Large-Scale Program Behavior Analysis for Adaptation and Parallelization

by

Xipeng Shen

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Chen Ding

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2006

To my dear wife, Xiaoyan, for giving me the greatest love and unconditional support.

Curriculum Vitae

Xipeng Shen was born in Shandong, China on April 20th, 1977. He attended the North China University of Technology from 1994 to 1998 receiving a Bachelor of Engineering in Industry Automation in 1998 and joined the Chinese Academy of Sciences after that. In 2001, he received his Master of Science in Pattern Recognition and Intelligent Systems. He came to the University of Rochester in the Fall of 2001 and began graduate studies in Computer Science. He pursued his research in programming systems under the direction of Professor Chen Ding. He received a Master of Science degree in Computer Science from the University of Rochester in 2003.

Acknowledgments

I have been very fortunate to have an advisor, Professor Chen Ding, who has provided me the independence to pursue my research interests and guided me through the course of this dissertation. From him, I learned not only the splendid attractions of science, a rigorous attitude toward research, and intelligent ways to solve problems, but also persistency in pursuing excellence, optimism when facing difficulties, and a profound understanding of life. Without his unconditional support and consistent inspirations, this thesis would not be possible.

I owe a particular debt to my committee. Sandhya Dwarkadas and Michael Scott have always promptly replied to my requests and put the greatest effort to provide me the excellent advice on my research, presentations and paper writings. Dana Ballard gave me invaluable encouragement when I was facing the difficulty of choosing my research area. Michael Huang offered insightful technical discussions and suggestions. Their supports helped to shape many aspects of this thesis.

In addition to my committee, Professor Kai Shen and Mitsunori Ogihara helped me with intelligent discussions on research and generous mentoring on my career search. I also thank Dr. Kevin Stoodley, Yaoqing Gao, and Roch Archambault for precious help and collaborations in IBM Toronto Lab.

All the people in the Computer Science department made my graduate life more enjoyable and helped me in various ways. In particular, I thank my collaborators, Yutao Zhong, Chengliang Zhang, Ruke Huang, Jonathan Shaw, Kirk Kelsey and Tongxin Bai for pleasant discussions.

I would express my special acknowledgment to my family. Mom and Dad gave me their greatest love, hope and belief. I would, above all, thank most my wife Xiaoyan for all she has brought me, the support, the love, and the best gift in the world—my dear son Daniel.

This material is based upon work supported by the National Science Foundation (Grants nos. CNS-0509270, CCR-0238176, CCR-0219848 and EIA-0080124) and the Department of Energy (Contract No. DE-FG02-02ER25525). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

Abstract

Motivated by the relentless quest for program performance and energy savings, program execution environments (e.g. computer architecture and operating systems) are becoming reconfigurable and adaptive. But most programs are not: despite dramatic differences in inputs, machine configurations, and the workload of the underlying operating systems, most programs always have the same code running with the same data structure. The resulting mismatch between program and environment often leads to execution slowdown and resource under-utilization. The problem is exacerbated as chip multi-processors are becoming commonplace and most user programs are still sequential, increasingly composed with library code and running with interpreters and virtual machines. The ultimate goal of my research is an intelligent programming system, which injects into a program the ability to automatically adapt and evolve its code and data and configure its running environment in order to achieve a better match between the (improved) program, its input, and the environment.

Program adaptation is not possible without accurately forecasting a program's behavior. However, traditional modular program design and analysis are ill-fitted for finding large-scale composite patterns in increasingly complicated code, dynamically allocated data, and multi-layered execution environments (e.g. interpreters, virtual machines, operating systems and computer architecture.) My research views a program as a composition of large-scale behavior patterns, each of which may span a large number of loops and procedures statically and billions of instructions dynamically. I apply statistical technology to automatically recognize the patterns, build models of program

behavior, and exploit them in offline program transformation (e.g. array regrouping based on *locality and reference affinity research*) and online program adaptation (e.g. *behavior-oriented parallelization* based on *behavior phases*) to improve program performance and reliability.

Table of Contents

List of Tables

List of Figures

1 Introduction

The relentless quest for performance motivated the fast development of computers in the past decades. Although processors have been following Moore's Law and doubled the speed every two years, users are demanding more: in physics, to study the slip condition, it takes hundreds of days to simulate fluid in channels; for computer architecture research, it takes about 60 hours to simulate one-minute program execution; for general users, the amount of data in the world is doubling every three years [Chen et al., 2005]. Despite the urgency for performance, power and heat constraints have stopped the increase of CPU speed. Instead, computers are increasingly equipped with more processors. How can a programming system further accelerate programs' execution and help users effectively utilize the extra computing resource?

This thesis presents a novel technique, *behavior-based program analysis*, to systematically characterize and predict large-scale program behavior. Unlike previous program analysis, we model program behavior as a composite effect from code, data, input and running environment rather than study them separately. The analysis builds a regression model to better predict whole-program locality across inputs. It develops a lightweight approach to recognize data affinity groups and improves program locality. For run-time adaptation, we propose behavior phases to capture large-scale dynamic behavior patterns. To ease the development of parallel programs, we construct a behavior-oriented parallelization system to (semi-)automatically parallelize complex programs. All the techniques are based on our distinguishing view of program behavior, which leads to the definition of our program behavior model.

1.1 Program Behavior Model

Behavior refers to “the actions or reactions of an object or organism, usually in relation to the environment ” (adapted from [Merriam-Webster, 1998].) In program world an “object ” is a program, including its code and data organization (data structure and data layout in the memory); the “environment” is program inputs and running context, including the situations of hardware, operating systems and virtual machines; an “action” is a program operation (in different scales) and the ensuing activities of the computing system like instruction execution, memory accesses and disk operations. Formally, we define program behavior as follows:

> **Program behavior** refers to the operations of a program—code and data— and the ensuing activities of the computing system on different scales in relation to the input and running environment.

The definition implies that *program behavior is a composite multiscale effect from program code, dynamic data, input, and running environment*, as denoted by the following formula. In short, we call those factors *program behavior components*.

> **behavior = code + dynamic data + input + environment + scale**

Program behavior has different granularities. It could be as small as loading a single data, or as large as the execution of the whole program. This thesis focuses on large-scale behavior: the locality and reference affinity models characterize whole-program behavior; phase analysis explores recurring and predictable program segments, which often span millions or billions of dynamic instructions and are not constrained by either program code structures or a fixed window size.

There are three reasons for focusing on large-scale behavior. First, it has more significant influence on the whole-program execution than fine-grain behavior. Second, it is not as sensitive to the randomness of the running environment as small-scale behavior, thus more regularity and better predicability. Last but not least, large granularity allows more sophisticated dynamic optimizations thanks to better tolerance of overhead.

1.2 New Challenges to Programming Systems

The mismatch among program behavior components, such as a memory layout with temporally close data being spatially far or a parallelizable application sequentially running on a parallel machine, usually causes program slowdown but computing resource under-utilized. It is the task of programming systems to "understand" a program, capturing behavior patterns and modeling the relations among behavior components, and then transform the program for a better match.

Prior programming systems fall into four categories, systems relying on *static analysis*, *offline profiling*, *run-time analysis*, or *hybrid approaches*. Static program analysis focuses on program code and has no knowledge of program input, running environment and thus run-time behavior. The analysis is therefore conservative, capturing only some static properties [Allen and Kennedy, 2001; Cooper and Torczon, 2004]. Techniques of offline profiling run a program with some training input and analyze that particular run to optimize the program (e.g. [Thabit, 1981]). Many kinds of behavior are input-sensitive, making the patterns learned from the training run unfit for other runs. Run-time analysis, conducted during a program's execution, is able to measure the most accurate run-time behavior, but cannot afford intensive analysis and large-scale transformations. There are some hybrid systems (e.g. [Grant et al., 1999a; Voss and Eigenmann, 2001]), using compiler or offline-profiling analysis to pre-plan for more efficient run-time optimizations. But due to the lack of large-scale behavior models, those techniques are limited in granularity and effectiveness.

The recent hardware and software trends exasperate the difficulties. In 1965, Intel co-founder Gordon Moore made the prediction, popularly known as Moore's Law, that the number of transistors on a chip doubles about every two years. Processor manufactures have been following Moore's law and keeping silicon integration in the last decades. However, the power leakage becomes a serious problem as gate oxide layers are becoming only several atoms thick. The power consumption and heat problem make higher frequency very difficult, forcing processor manufactures to change their direction from increasing clock frequency to increasing concurrency. Technology trends also show that global on-chip wire delays are growing significantly, eventually increasing cross-chip communication latencies to tens of cycles and rendering the expected chip area reachable in a single cycle to be less than 1% in a 35nm technology [Keckler et al., 2003]. In the future, computers will therefore be equipped with more cores per chip rather than faster processors. The different intra- and intel-chip wire delays require different granularities of concurrency, which raises the urgency for programming systems to better understand programs and capture behavior patterns (e.g. locality and concurrency) of different scales.

Another trend in hardware is more flexibility. To save energy, the voltage is becoming scalable [Burd and Brodersen, 1995] and cache systems are becoming reconfigurable [Balasubramonian et al., 2000b]. Guiding those reconfigurations is critical to effectively exploit the flexibility, which again relies on the accurate prediction of program large-scale behavior.

On the other hand, modern software is becoming more difficult to analyze. Programs are becoming more complex and increasingly composed of codes from libraries and third parties; programmers are embracing high-level object-oriented languages such as Java and C# due to their software engineering benefits. These programs use small methods, dynamic class binding, heavy memory allocation, short-lived objects, and pointer data structures, and thus obscure parallelism, locality, and control flow, in direct conflict with hardware trends [McKinley, 2004]. Furthermore, those programs run on a multi-layered environment composed of interpreters, virtual machines, operating systems and hardware.

The opposite trends of software and hardware implies increasing urgencies and challenges for programming systems to better match behavior components.

This thesis presents behavior-based program analysis to systematically model and predict the effects of various components on program large-scale behavior. It demonstrates the effectiveness in program adaptation and automatic parallelization.

1.3 Behavior-Based Program Analysis

Behavior-based program analysis focuses on dynamic composite behavior and builds statistical models to link large-scale program behavior with its code, data organization, input and execution environment. It bridges static analysis, profiling-based analysis and runtime analysis, providing a new way to capture large-scale program dynamic behavior patterns. My research focuses on program memory behavior because of the widening speedup gap between CPU and DRAM as shown in Figure 1.1 [Hennessy and Patterson, 2003]. The gap results in a severe performance bottleneck in modern computers. This thesis starts from the exploration of program average behavior to dynamic behavior, spanning offline program optimizations as well as runtime adaptation and automatic parallelization.

Data Locality

Locality or data reuse determines the effectiveness of caching, an important factor on system performance, cost, and energy usage, especially as new cache designs are adding more levels and dynamic reconfiguration. My research starts from modeling the connection between program input and *whole-program locality*, defined as the distribution of data reuse distance in a program execution. The thesis presents a mixture model

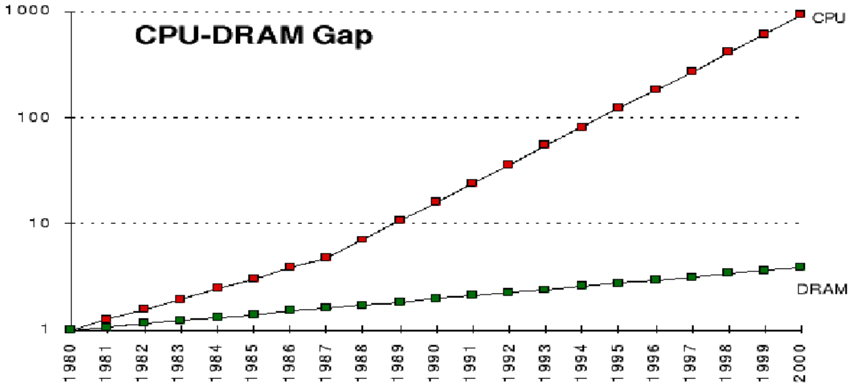


Figure 1.1: The speedup curves of CPU and DRAM from 1980 to 2000 showed by J. Hennessy and D. Goldberg.

with regression analysis to predict whole-program locality for all inputs. The new approach is distinguished from previous work in two aspects. First, the model relaxes the "pure model per group" assumption by allowing mixed data reuse patterns. That makes small inputs enough for locality analysis and consequently reduces the majority of the training overhead. Second, regression analysis improves locality prediction accuracy by taking advantage of more than two training inputs. Compared to previous methods, the new locality prediction reduces about half of the prediction error, removes 95% of space cost, and uses much smaller inputs and faster data collection [Shen et al., 2003]. The locality analysis has been used to generate a parameterized model of program cache behavior. Given a cache size and associativity, the model predicts the miss rate for an arbitrary data input. It also identifies critical data input sizes where cache behavior exhibits marked changes. Experiments show this technique is within 2% of the hit rates for set associative caches on a set of floating-point and integer programs using array- and pointer-based data structures [Zhong et al., To appear]. (The miss rate error can be larger especially for low miss rates.) The model enables better understanding of program cache behavior, helps machine and benchmark design, and assists reconfigurable memory systems.

Reference Affinity

While the memory of most machines is organized as a hierarchy, program data are laid out in a uniform address space. Data reorganization is critical for improving cache spatial locality and thus memory bandwidth. Array regrouping, for example, combines arrays that are often accessed together into a structure array so that a single cache load operation can load the elements from multiple arrays residing on the same cache line after regrouping. We have proposed *reference affinity*, the first trace-based model of hierarchical data locality, and a distance-based affinity analysis, which finds array and structure field organization (among an exponential number of choices) that is consistently better than the layout given by the programmer, compiler, or statistical clustering [Zhong et al., 2004]. To make the affinity model practical for general-purpose compilers, this thesis proposes a lightweight affinity model based on static interprocedure analysis. The prototype has been implemented in the IBM C/C++ and FORTRAN production compiler. Both techniques lead to significant performance improvements, up to 96% (on average 30%) speedup of a set of SPEC 2000 floating-point benchmarks [Shen et al., 2005].

The second part of the thesis focuses on dynamic memory behavior patterns for online program adaptation and parallelization.

Behavior Phases

While whole-program models give the average behavior, our research in program phases goes one step further to model and predict dynamic behavior changes at run time. We use multiple training runs to statistically identify large-scale behavior patterns, which we call *behavior phases*, and exploits phases to guide online program adaptation to improve cache performance, better memory management, and increase parallelism.

Many programs, e.g. scientific simulation programs, have long continuous phases of execution that have dynamic but predictable locality. To support phased-based memory adaptation (e.g. reorganizing data for different phases), this thesis presents a novel technique which applies signal processing technique, wavelet transform, to identify phases from billions of data accesses. Frequency-based phase marking is then used to insert code markers that mark phases in all executions of the program. Phase hierarchy construction identifies the structure of all phases through grammar compression. With phase markers inserted, the run-time system uses the first few executions of a phase to predict all its later executions. The technique shows effectiveness in guiding cache resizing in reconfigurable systems, where the cache size can be adjusted for energy and performance, and memory remapping, where data can be reorganized at phase boundaries [Shen et al., 2004b,c].

Outside the realm of scientific computing, many programs, such as programming tools, server applications, databases, and interpreters, produce (nearly) identical service to a sequence of requests. Those programs typically use dynamic data and control structures and reveal different behavior for different requests, which can easily hide those aspects of behavior that are uniform across inputs and make it difficult or impossible for current analysis to predict run-time behavior. We call those programs *utility programs*. The repetitive behavior of utility programs, while often clear to users, has been difficult to capture automatically. This thesis proposes an *active profiling* technique, which exploits controlled inputs to trigger regular behavior and then recognizes and inserts common phase markers through profiling runs on normal inputs. Because users control the selection of regular inputs, active profiling can also be used to build specialized versions of utility programs for different purposes, breaking away from the traditional “one-binary-fits-all” program model. Experiments with five utilities from SPEC benchmark suites show that phase behavior is surprisingly predictable in many (though not all) cases. This predictability can in turn be used for better memory management including preventive garbage collection (to invoke garbage collection at phase boundaries which usually correspond to memory-usage boundaries), memory-usage monitoring (to better predict memory usage trend through monitoring at phase boundaries), and memory leak detection (to detect potential memory leaks by identifying phase-local objects), leading in several cases to multi-fold improvements in performance [Shen et al., 2004a; Ding et al., 2005; Zhang et al., 2006].

Behavior-Oriented Parallelization

Adaptive profiling also suggests the possibility of automatic coarse-grain parallelization, a special kind of program adaptation. Many programs have high-level parallelism that users understand well at the behavioral level. Examples include a file compressor compressing a set of files, a natural language parser parsing many sentences, and other utility programs. These programs are widely used and important to parallelize as desktop and workstations are increasingly equipped with chip multi-processors.

High-level parallelization of utility programs is difficult for traditional compiler analysis as the large parallelization region may span many functions with complicated control-flow and dynamic data accesses. Furthermore, the parallelism is often dynamic and input dependent: depending on the input, there may be full, partial, or no parallelism in an execution.

This thesis presents behavior-oriented parallelization (BOP), which is adaptive to program inputs and run-time behavior, but relies on no special hardware. It has three components. The first is training-based behavior analysis, which identifies the recurring phases in a utility program, measures the parallelism among phase instances, and analyzes the control flow and data structures for possible parallelization. The second is behavior speculation, which transforms the program so it speculatively executes later phase instances, checks the correctness, jumps forward if speculation succeeds, but cancels the effects if speculation fails. Finally, while training-based analysis identifies likely behavior, a behavior support system ensures correct and efficient execution for all behavior. Experiments on two commonly used open-source applications demonstrate as much as 78% speedup on a dual-CPU machine [Shen and Ding, 2005].

1.4 Dissertation Organization

The dissertation is organized as follows. Chapter 2 focuses on whole-program locality prediction. It presents a regression model to predict whole-program locality across inputs. Chapter 3 discusses the reference affinity model and proposes a lightweight affinity analysis to determine data affinity groups and improve program locality. Both chapters are devoted to the average behavior of a whole program. Chapter 4 starts the study of program behavior phases by presenting a wavelet transform based analysis to detect and predict large-scale program phase behavior. Chapter 5 extends the phase analysis to input-sensitive utility programs through active profiling. Chapter 6 focuses on a framework to (semi-)automatically parallelize complex integer programs through the support of a modified C compiler, offline profiling, and a runtime system. Chapter 7 summarizes the contributions and discusses possible extensions.

2 Data Locality

Locality is critical for amortizing the memory bottleneck due to the increasing CPU-memory speed gap. As a locality metric, the distance of data reuses has been used in designing compiler, architecture, and file systems. Data reuse behavior is input-sensitive: two executions with different input often have significantly different data reuse patterns. To better understand locality across program inputs, Ding and Zhong proposed a method to build linear model using two training runs, which for the first time enables the prediction of reuse distance histograms of the execution on an arbitrary input [Ding and Zhong, 2003]. This chapter discusses the technique and presents a set of new methods with two extensions. First is the regression analysis on more than two training inputs. Second is a multi-model technique to reduce prediction errors due to small training inputs or coarse-grain data collection. The new locality prediction improves accuracy for 50%, removes 95% of space cost, and uses much smaller inputs and thus much faster data collection in model training.

2.1 Introduction

Caching is widely used in many computer programs and systems, and cache performance increasingly determines system speed, cost, and energy usage. The effect of caching is determined by the locality of the memory access of a program. As new cache designs are adding more cache levels and dynamic reconfiguration schemes, cache performance increasingly depends on the ability to predict program locality.

Many programs have predictable data-access patterns. Some patterns change from one input to another, for example, a finite-element analysis for different size terrains and a Fourier transformation for different length signals. Some patterns are constant, for example, a chess program looking ahead a finite number of moves and a compression tool operating over a constant-size window.

The past work provides mainly three ways of locality analysis: by a compiler, which analyzes loop nests but is not as effective for dynamic control flow and data indirection; by a profiler, which analyzes a program for select inputs but does not predict its behavior change in other inputs; or by run-time analysis, which cannot afford to analyze every access to every data. The inquiry continues for a prediction scheme that is efficient, accurate, and applicable to general-purpose programs.

Ding and Zhong [Ding and Zhong, 2003] presents a method for locality prediction across program inputs, using a concept called the *reuse distance*. In a sequential execution, the **reuse distance** of a data access is the number of *distinct* data elements accessed between this and the previous access of the same data. It is the same as *LRU stack distance* proposed by Mattson et al. [Mattson et al., 1970].

Ding and Zhong describe three properties of the reuse distance that are critical for predicting program locality across different executions of a program. First, the reuse distance is at most a linear function of the program data size. The search space is much smaller for pattern recognition and prediction. Second, the reuse distance reveals invariance in program behavior. Most control flow perturbs only short access sequences but not the cumulative distance over a large amount of data. Long reuse distances suggest important data and signal major phases of a program. Finally, reuse distance allows direct comparison of data behavior in different program runs. Different executions of a program may allocate different data or allocate the same data at different locations. They may go through different paths. Distance-based correlation does not require two executions to have the same data or to execute the same functions. Therefore, it can identify consistent patterns in the presence of dynamic data allocation and input-dependent control flows [Ding and Zhong, 2003].

Ding and Zhong show that the histogram of reuse distance, also called *reuse signature*, of many programs has a consistent pattern across different inputs. The pattern is a parameterized formula that for a given program input, it predicts the reuse signature for the corresponding execution.

However, their pattern analysis has two limitations. First, it uses only two training runs and therefore may be misled by noises from specific executions. Second, the accuracy is limited by the precision of data collection. Accurate prediction requires large size program inputs and fine-grained reuse distance histograms. The space and time cost of the analysis is consequently high, which makes the analysis slower and prohibits simultaneous analysis of different patterns, for example, patterns of individual data elements.

This chapter presents a new set of techniques that overcome these limitations in two ways. First, we use regression to extract signature patterns from more than two training runs. Second, we employ multi-models.

2.1.1 Basic Prediction Method

This section describes the basic locality prediction approach and the main factors affecting the prediction accuracy.

Given an input to a program, we measure the locality of the execution by the histogram of the distance of all data reuses also called *reuse distance histogram* or *reuse signature* (see Section 2.2 for formal definitions). The prediction method by Ding and Zhong uses a training step to construct a pattern by running two different inputs of a program. Let s and $\hat{s}$ be the sizes of the two input data. For each of the reuse distance histogram, the analysis forms 1000 groups by assigning 0.1% of all memory accesses to each group, starting from the shortest reuse distance to the largest. We denote the two sets of 1000 groups as $\langle g_1, g_2, \cdots, g_{1000}\rangle$ and $\langle \hat{g}_1, \hat{g}_2, \cdots, \hat{g}_{1000}\rangle$ and denote the average reuse distances of g_i and $\hat{g}_i$ by rd_i and $\hat{rd}_i$ respectively ($i = 1, 2, \cdots, 1000$.) Based on rd_i and $\hat{rd}_i$, the analysis classifies group i as a constant, linear, or sub-linear pattern. Group i has a constant pattern if its average reuse distance stays the same in the two runs, i.e. $rd_i = \hat{rd}_i$. Group i has a linear pattern if the average distance changes linearly with the change in program input size, i.e. $\frac{rd_i}{\hat{rd}_i} = c + k\frac{s}{\hat{s}}$, where c and k are both constant parameters. Ding and Zhong measured the size of input data through distance-based sampling [Ding and Zhong, 2003]. We use the same sampling method in this work.

After the training step, the reuse signature for another input can be predicted by calculating the new distance for each group according to its pattern. Interested reader can find a more detail discussion of this process in Ding and Zhong's paper [Ding and Zhong, 2003]. Figure 2.1 shows the flow diagram of their prediction method. We will explain the different types of histograms in Section 2.2.

Note that not all programs have a consistent pattern, and not all patterns are predictable. However, Ding and Zhong showed that their method can find predictable patterns in a wide range of large, complex programs. The goal of this work is to improve the analysis accuracy and efficiency for programs that have a predictable pattern.

2.1.2 Factors Affecting Prediction Accuracy

Three factors strongly affect the prediction accuracy: the number of training inputs, the precision of data collection, and the complexity of patterns. The number of training inputs needs to be at least two, although using more inputs may allow more precise recognition of common patterns. The precision of data collection is determined by the number of groups. Since each group is represented by its average reuse distance, the more groups the analysis uses, the more precise the reuse distance information is.

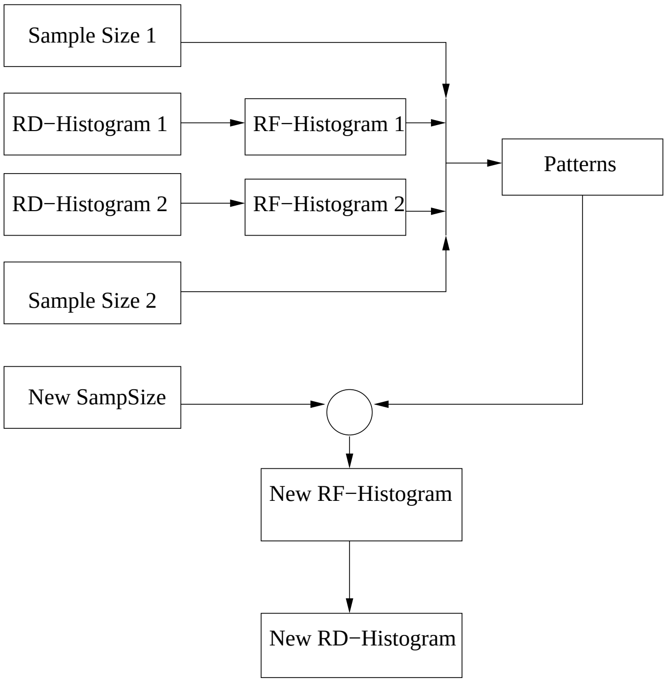


Figure 2.1: The flow diagram of Ding and Zhong's prediction method, which uses only two training inputs. A RD-Histogram is a reuse-distance histogram, and a RF-Histogram is a reference histogram. Sample size is the estimated input data size by sampling.

However, using more groups leads to slower pattern recognition and prediction since the space and time costs are proportional to the number of groups. The third factor is the complexity of patterns in each group. If we assume that the entire group has a single pattern, the analysis is a *single-model* prediction. If we assume that the group may consist of different subgroups that each may have a different pattern, the analysis is a *multi-model* prediction.

Single-model prediction has two limitations. First, the accuracy of the prediction is strictly limited by the precision of data collection, i.e. the number of groups. A large group tends to include subgroups with different patterns, which breaks the single-model assumption and causes low prediction accuracy. Second, training runs need to have a sufficient size so that the range of reuse distances in different patterns can be well separated. The larger the input data size is, the more likely different patterns is separated. If the distances of two patterns are similar, they fall into the same group, and the prediction cannot tear them apart. Because of the need for large training inputs and number of groups, single-model prediction usually incurs a large time and space cost. Multi-model prediction, however, may overcome these two limitations by allowing subportions of a group to have a different pattern.

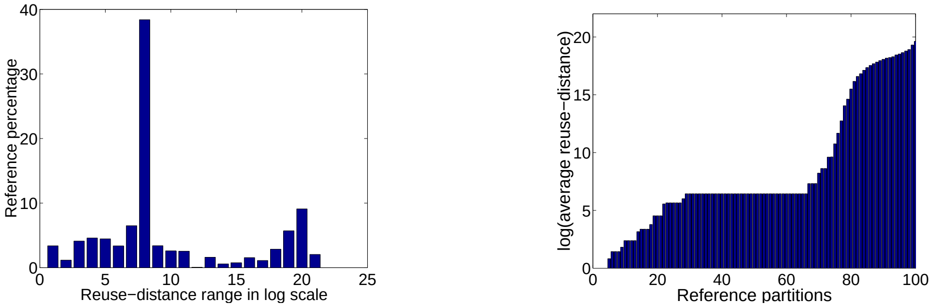
Our extension to their method has three contributions. The first is single-model prediction using more than two training runs. The next is a set of multi-model prediction methods using different types of reuse distance histograms. It reduces the space cost from $O(M)$ to $O(logM)$, where M is the size of program data. The last is a strategy for choosing the appropriate histograms based on analytical and experimental results.

The rest of the chapter is organized as follows. Section 2.2 describes the types of histograms used in our prediction. Section 2.3 and 2.4 describe the new regression-based multi-model methods. Followed are the experiment results and discussions. Section 2.7 discusses related work. Section 2.8 speculates the possible extensions, and Section 2.9 summarizes our findings.

2.2 Terminology

This section explains the two types of histograms and related terms used in later discussions.

- **A Reuse-distance Histogram (RD-Histogram):** the X-axis is reuse-distance ranges, and the Y-axis is the percentage of data accesses in each distance range. The size of distance ranges can be in linear scale, e.g. $[0, 1k), [1k, 2k)$, $[2k, 3k), \cdots$, or log scale, e.g. $[0, 1k), [1k, 2k), [2k, 4k), [4k, 8k), \cdots$, or mixed linear and log scales. Figure 2.2(a) shows the reuse-distance histogram of SP in log scale ranges.



(a) (b)

Figure 2.2: Reuse-distance histogram of *SP* with input of size 28^3. (a) distance histogram (b) reference histogram

- **A Reference Histogram (RF-Histogram):** the X-axis is the groups of data accesses, sorted by the average reuse distance. The Y-axis is the average reuse distance of each partition. Figure 2.2(b) is the reference histogram of SP for a partition of 100 groups. A reference histogram can be viewed as a special type of reuse-distance histogram whose distance ranges have non-uniform lengths so that each range holds the same number of program data accesses.

Reference histogram provides a trade-off between information loss and computation/space efficiency. For dense regions in the reuse distance histogram, where a large portion of memory accesses have similar reuse distances, the reference histogram uses short range to increase accuracy. For sparse regions in the reuse distance histogram, the reference histogram uses large ranges to reduce the total number of ranges.

2.3 Single-Model Multi-Input Prediction

Using more than two training inputs may reduce two kinds of noises and thus improve prediction accuracy. One kind of noise is brought by the reuse distance measurement. Ding and Zhong used approximation to trade accuracy for efficiency [Ding and

Zhong, 2003]. The approximation brings errors to the reuse distance histogram. The second kind of noise is the estimated data size from sampling. Although distance-based sampling [Ding and Zhong, 2003] finds a size reflecting the size of a program input, the sampled size is not always accurate. These noises reduce the accuracy of the prediction.

According to the regression theory, more data can reduce the effect of noises and reveal a pattern closer to the real pattern [Rawlings, 1988]. Accordingly, we apply a regression method on more than two training inputs. The extension is straightforward. For each input, we have an equation as follows.

$$d_i = c_i + e_i * f_i(s) \tag{2.1}$$

where d_i is the average reuse distance of i^{th} reference group when the input size is s, c_i and e_i are two parameters to be determined by the prediction method, and f_i is one of the following functions of s:

$$0; \;\; s; \;\; s^{1/2}; \;\; s^{1/3}; \;\; s^{2/3}$$

Given the histograms of two training runs, Ding and Zhong could solve a linear equation, determine the two unknowns for each group, and calculate the reuse distance histogram for a new input given its input size. Using two training inputs is sufficient because there are only two unknowns in each model (Equation 2.1).

While the previous method has two equations, we have more than two equations because of more training inputs. We use *least square regression* [Rawlings, 1988] to determine the best values for the two unknowns. We use 3 to 6 training inputs in our experiment. Although more training data can lead to better results, they also lengthen the profiling process. We will show that a small number of training inputs is sufficient to gain high prediction accuracy.

2.4 Multi-Model Prediction

A multi-model method assumes that memory accesses in a group can have different models. For example in a histogram, a bar (group) may contain both a constant model and a linear model.

Figure 2.3 gives a graphical illustration of the multi-model prediction. We arbitrarily pick one of the training inputs as the *standard input*. In this example, s_0 is the size of the standard input (the other training inputs are not showed in the figure.) Its reuse distance histogram, called *standard histogram*, has 12 groups, and each group consists of two models—constant and linear models. Together, they form the histogram of s_0. Using regression technique on all training histograms, the standard histogram in Figure 2.3(a) is decomposed into constant and linear models in Figure 2.3(b) and (c). The decomposition process is described below. For prediction, the two histograms become Figure 2.3(d) and (e) respectively according to the size of the new input $8 * s_0$. Constant histogram keeps unchanged, and the distance of each data in linear histogram increases to 8 times long. The X-axis is in log scale, so each bar in linear histogram moves 3 ranges right-toward. The reuse distance histogram for the new input is the combination of the new constant and linear histograms, see Figure 2.3(f).

Formally, the reuse distance function of a group is as follows.

$$h_i(s) = \varphi_{m_1}(s, i) + \varphi_{m_2}(s, i) + \cdots + \varphi_{m_j}(s, i) \tag{2.2}$$

where, s is the size of input data, $h_i(s)$ is the Y-axis value of the i^{th} bar/group for input of size s, and $\varphi_{m_1} ... \varphi_{m_j}$ are the functions corresponding to all possible models.

Each $h_i(s)$ can be represented as a linear combination of all the possible models of the standard histogram:

$$\varphi_{m_1}(s_0, 1), \varphi_{m_1}(s_0, 2), \cdots, \varphi_{m_1}(s_0, G), \varphi_{m_2}(s_0, 1), \varphi_{m_2}(s_0, 2), \cdots, \varphi_{m_2}(s_0, G),$$
$$\cdots, \varphi_{m_j}(s_0, 1), \varphi_{m_j}(s_0, 2), \cdots, \varphi_{m_j}(s_0, G)$$

where, G is number of groups in standard histogram.

For example, a program has both constant and linear patterns. For easy description, we assume the following ranges.

range 0: [0,1); range 1: [1,2); range 2: [2,4); range 3: [4,8), $\cdots$

For another input of size $s_1 = 3 * s_0$, we calculate the Y-axis value of range $[4, 8)$ as follows:

$$h_3(s_1) = \varphi_0(s_0, 3) + \varphi_1(s_0, r)$$

where, r is range $[\frac{4}{3}, \frac{8}{3})$. This is because the constant model of range $[4, 8)$ in the standard histogram gives entire contribution, and the linear model of $h_3(s_1)$ comes from the linear portions in range $[\frac{4}{3}, \frac{8}{3})$ of standard histogram. $\varphi_1(s_0, r)$ can be calculated as follows:

$$\varphi_1(s_0, r) = \varphi_1(s_0, r_1) + \varphi_1(s_0, r_2)$$

where, $r_1 = [\frac{4}{3}, 2)$ and $r_2 = [2, \frac{8}{3})$.

We assume the reuse distance has uniform distribution in each range. Thus,

$$\varphi_1(s_0, r_1) = (\tfrac{2-4/3}{2-1})\varphi_1(s_0, 1) = \tfrac{2}{3}\varphi_1(s_0, 1)$$
$$\varphi_1(s_0, r_2) = (\tfrac{8/3-2}{4-2})\varphi_1(s_0, 2) = \tfrac{1}{3}\varphi_1(s_0, 2)$$

Finally, we calculate $h_3(s_1)$ as follows:

$$h_3(s_1) = \varphi_0(s_0, 3) + \tfrac{2}{3}\varphi_1(s_0, 1) + \tfrac{1}{3}\varphi_1(s_0, 2)$$

After we represent each $h_i(s)$ of all training inputs in the above manner, we obtain an equation group. The unknown variables are the models in the standard histogram. The equations correspond to the groups in all training histograms. Regression techniques can solve the equation group. This completes the decomposition process and also completes the construction of reuse distance predictor. During prediction process, for any input, each of its reuse distance group can be calculated as a linear combination of standard histogram models in the same manner as in decomposition process. Then, its reuse distance histogram can be obtained by the combination of all the groups.

One important assumption is that the percentage of memory accesses in each model remains unchanged for different inputs. There is no guarantee that this is the case, although Ding and Zhong showed that it is an acceptable assumption for a range of programs including those used in this work.

A multi-model method does not depend on the type of histograms. It can use distance histograms with log or $linear$ size groups. It can also use reference histograms. The equations are constructed and solved in the same manner.

We now describe three methods of multi-model prediction. They differ by the type of reuse distance histograms. The first two methods use reuse distance histograms with log and log-$linear$ scale ranges respectively. The first 13 ranges in the log-$linear$ scale is in log scale (power-of-two) and the rest have length 2048. The purpose of the log part is to distinguish groups with short reuse distances. The space and time cost of the second method is $O(M)$, where M is the size of program data in training runs. The space and time cost of the first method is $O(logM)$, which saves significant space and time because it has much fewer equations and variables. However, the linear scale has higher precision, which can produce better results especially when using small size training runs.

The third multi-model method uses a reference histogram, for example, with 1000 groups. Unlike the first two methods, in this method, the number of equations and variables is the same as the number of groups. We can choose an arbitrary number. This provides freedom but also raises a problem: how to choose the best number of groups. In fact, the last method represents as many methods as the maximal number of groups, which is $O(N)$, where N is the number of memory accesses in the smallest training run. We will see in the evaluation section that the prediction accuracy depends heavily on the choice of groups.

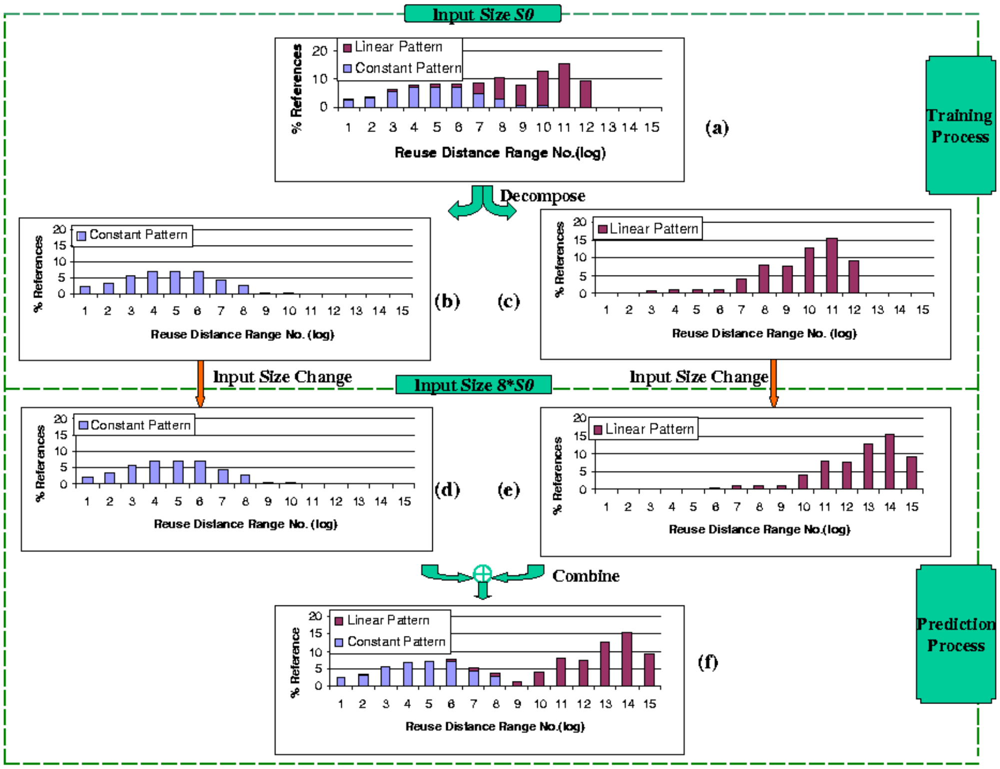


Figure 2.3: An example for multi-model reuse signature prediction. Figure (a) is the reuse distance histogram of the execution on standard input s_0. By using regression technique on all training histograms, the standard histogram is decomposed into two histograms—constant and linear histograms in Figure (b) and (c). During the prediction process, the two histograms become Figure (d) and (e) respectively according to the size of the new input $8 * s_0$. The constant histogram remains unchanged, and the distance of each data in a linear histogram increases to 8 times long. The X-axis is in log scale, so each bar of linear pattern moves 3 ranges right-toward. The reuse distance histogram for the new input is the combination of the new constant and linear histograms, showed in Figure (f).

2.5 Evaluation

2.5.1 Experimental Setup

We compare five prediction methods: the single-model two-input method given by Ding and Zhong, the single-model multi-input regression described in Section 2.3, and the three multi-model methods described in Section 2.4. The multi-input methods use

Table 2.1: Benchmarks for locality prediction

Benchmark	Description	Suite
Applu	solution of five coupled nonlinear PDE's	Spec2K
SP	computational fluid dynamics (CFD) simulation	NAS
FFT	fast Fourier transformation	
Tomcatv	vectorized mesh generation	Spec95
GCC	based on the GNU C compiler version 2.5.3	Spec95
Swim	finite difference approximations for shallow water equation	Spec95

3 to 6 training inputs. We measure accuracy by comparing the predicted histogram with the measured histogram for a test input. The definition of accuracy is the same as Ding and Zhong's [Ding and Zhong, 2003]. Let x_i and y_i be the size of ith groups in the predicted and measured histograms. The cumulative difference, E, is the sum of $\mid y_i - x_i \mid$ for all i. The accuracy A is $(1 - E/2)$, which intuitively is the overlap between the two histograms.

Table 2.1 lists the names of six test programs, their descriptions, and the sources. Table 2.2 and Figure 2.4 show the accuracy of the five approaches on six benchmarks when training and testing inputs are large. In the table, "Max. Inputs Num." is the maximal number of inputs among all the five methods for each benchmark. In our experiment, for each benchmark, the size of the biggest training input is the same for all five methods. This is to make the comparison fair.

2.5.2 Results on Large Inputs

Using a large input has two benefits. First, different models stay separated from each other. For example, suppose constant and linear models co-exist in a range r when the input size is s_0. For a larger input whose size is $1024 * s_0$, the linear model will move far out of range r, but constant model remains unchanged. Thus, there will not be an overlap of models. The separation of models is important for the two single-model methods since they assume that only one model exists in each range. The second

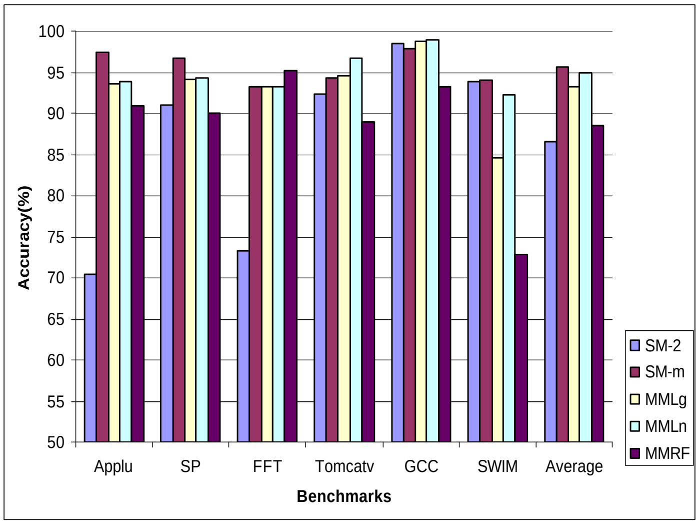


Figure 2.4: Locality prediction accuracy bar graph.

benefit is that the percentage of individual models is more likely to remain constant when the input size is large. This is required by both single-model and multi-model based methods.

Table 2.2: Comparison of prediction accuracy by five methods

Benchmark	Single Model		Multi-model			Max. Inputs Num.∗
	2 inputs RF-Hist.	>2 inputs RF-Hist.	Log RD-Hist.	log-Linear RD-Hist.	Fixed RF-Hist.	
Applu	70.49	97.40	93.65	93.90	90.83	6
SP	91.08	96.69	94.20	94.37	90.02	5
FFT	73.28	93.30	93.22	93.34	95.26	3
Tomcatv	92.32	94.38	94.70	96.69	88.89	5
GCC	98.50	97.95	98.83	98.91	93.34	4
SWIM	93.89	94.05	84.67	92.20	72.84	5
Average	**86.59**	**95.63**	**93.21**	**94.90**	**88.53**	**4.7**

The first column of Table 2.2 gives the results of Ding and Zhong's method. Other columns show the accuracy of the new methods. All methods are based on histograms given by the same reuse-distance analyzer and the input sizes given by the same distance-based sampler. The numbers of the first column is slightly different from Ding and Zhong's paper [Ding and Zhong, 2003], because they used a different reuse-distance analyzer than we do. Different analyzers lose precision in slightly different ways. The sampled input size is also slightly different because of this. From Table 2.2, we make the following observations:

- For most programs, all four new approaches produce better results than Ding and Zhong's method. Therefore, regression on multiple inputs indeed improves prediction accuracy.
- Except for *SWIM*, multi-model logarithmic scale method is comparable to the best predictors, although it uses 95% less storage space in most analysis. It is the most efficient among all methods.
- The performance of multi-model log-linear scale method is slightly better than multi-model logarithmic scale method for the first four benchmarks and much better for *SWIM*. However, log-linear scale costs more than 20 times in space and computations than logarithmic scale for most programs.
- The multi-model method based on reference histograms outperforms single-model two-input method for two out of six programs. It gives the highest accuracy for FFT. As we explained in Section 2.4, this approach is very flexible and its performance depends heavily on the number of groups. In our experiment, we tried 7 different numbers of groups for each benchmark and presented the highest accuracy, but finding the maximal accuracy requires trying thousands of choices. The result for FFT shows the potential of this method, but the overhead of finding the best result is prohibitively high.

SWIM is a special program. The multi-model logarithmic scale has poor result for *SWIM*, but multi-model log-linear scale and single-model methods give very accurate predictions. Figure 2.5 shows the reuse distance histogram of *SWIM*. Note it has a high

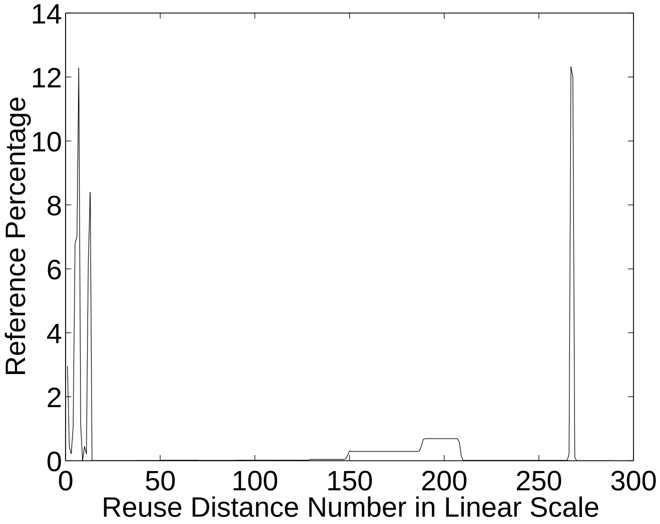


Figure 2.5: The reuse distance histogram curve of *SWIM*

peak in a very small reuse distance range. Multi-model logarithmic scale uses log scale ranges. It assumes that the reuse distance is evenly distributed in each range, which brings significant noise for the analysis of *SWIM*. Log-linear scale methods alleviate the problem because their histograms are much more precise.

2.5.3 Results on Small Inputs

As we explained at the beginning of Section 2.5.2, different patterns may overlap with each other when the input size is small. In this case, single-model methods are not expected to perform well, while multi-model methods should work as well as in large input sizes. But these methods still require that the percentage of each model keeps unchanged for different input for each reuse distance range. Table 2.3 shows the performance of the four methods on small size inputs of SP benchmark (We do not show the results of the multi-model method using reference histograms because it is difficult to tune). The results show that multi-model log-linear scale method is significantly more accurate than other methods. The good accuracy shows that the percentage of each model remains unchanged even for small inputs. The performance of multi-model logarithmic scale method is worse than the log-linear scale method because of the low precision in logarithmic scale histograms. Although multi-model log-linear scale method needs more computation and more space than the logarithmic scale method, this cost is less an issue for small-size inputs.

Table 2.3: Accuracy for SP with small-size inputs

largest training input size	testing input size	single-model 2 inputs	single-model $>$2 inputs	multi-model log scale	multi-model log-linear scale
8^3	10^3	79.61	79.61	85.92	89.5
	12^3	79.72	75.93	79.35	82.84
	14^3	69.62	71.12	74.12	85.14
	28^3	64.38	68.03	76.46	80.3
10^3	12^3	91.25	87.09	84.58	90.44
	14^3	81.91	83.20	78.52	87.23
	16^3	77.28	77.64	76.01	84.61
16^3	28^3	75.93	74.11	77.86	83.50

2.5.4 Comparison

We compare the five methods in Table 2.4, which uses the following notations:

SM-2: Ding and Zhong’s original method

SM-m: Extended version of SM-2 on multiple inputs

MMLg: Multi-model *log* scale method

MMLn: Multi-model log-linear scale method

MMRF: Multi-model on reference histogram

A comparison of the four new approaches is as follows.

- *SM-m* and *MMRF* are based on reference histograms while *MMLg* and *MMLn* are based on reuse distance histograms. Thus, *MMLg* and *MMLn* do not need to transform between the two histograms but *SM-m* and *MMRF* do.

Table 2.4: Features of various reuse distance prediction methods

Approach	*SM-2*	*SM-m*	*MMLg*	*MMLn*	*MMRF*
Input No.	2	>2	>2	>2	>2
Model No. per Range	1	1	≥1	≥1	≥1
Histogram	Ref.	Ref.	Dist.	Dist.	Ref.
Granularity	log-linear	log-linear	log	log-linear	log-linear

- *MMLg* saves 20 times in space and computation compared to *SM-m* and *MMLn*. *MMRF* can also save cost because it can freely select the number of groups, but it is hard to pick the right number.

- *MMLg* loses information because it assumes a uniform distribution in large ranges. That hurts the prediction accuracy for programs like $SWIM$, which has a high peak in a very small range. In that case, *SM-m* and *MMLn* produce much better results because they use shorter ranges.

- *MMLn* predicts with higher accuracy than *SM-m* does if multiple models overlap. Overlapping often happens for inputs of small size, for which *MMLg* cannot perform well because of its loss of information.

Summary The experiments show that regression on more than two training inputs gives significant improvement compared to the method using two inputs. Single-model multi-input, and multi-model logarithmic and log-linear scale methods produce comparable results for most programs when the input size is large. Their overall performance is the best among all five approaches. The multi-model method using reference histograms method is flexible but hard to control. Multi-model log-linear scale method can produce better results than multi-model logarithmic scale method. But the former needs over 20 times of more space and time than the latter, and the performance is not significantly different in most programs when the input size is large. For input of small size, the log-linear scale method is clearly the best among all methods.

2.6 Uses in Cache Performance Prediction

Prior work on cache characterization addresses how to quickly explore the planar space delineated by the size and associativity. Locality prediction across inputs extends the exploration along the program data set size dimension in an efficient manner [Zhong et al., To appear, 2003a].

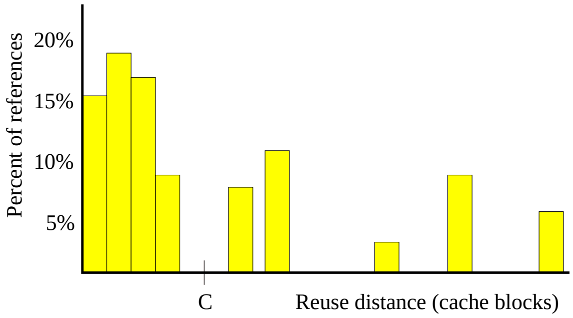


Figure 2.6: Reuse distance histogram example

Treating each distinct cache block as a basic data element, we use the regression technique to build the cache reuse distance histogram model. For an arbitrary input, the new cache reuse distance histogram can be predicted as illustrated by Figure 2.6. The number of intervening cache blocks between two consecutive accesses to a cache block, along with cache size, determines whether the second access hits or misses in a fully associative LRU cache. This is the basis for our prediction model. In Figure 2.6, the fraction of references to the left of the mark C will hit in a fully associative cache having capacity of C blocks or greater. For set associative caches, reuse distance is still an important hint to cache behavior [Smith, 1978; Beyls and D'Hollander, 2001].

We use the cache simulator *Cheetah* [Sugumar and Abraham, 1991] included in SimpleScalar 3.0 toolset [Burger and Austin, 1997] to collect cache miss statistics. Cache configurations are fully associative cache, 1-, 2-, 4- and 8-ways, all having block size of 32 bytes. Our experiments on 10 benchmarks (from SPEC95, SPEC2k, NAS, and Olden suites) show that the prediction accuracy of cache hit rates is always higher than 99% for fully associative caches, and better than 98% for caches of limited associativity for all but two programs, excluding compulsory misses. In addition, the predicted miss rate is either very close or proportional to the miss rate of direct-map or set-associative cache.

2.7 Related Work

Locality prediction is one example to connect input with program behavior. Adaptive algorithms is another one. For example, we proposed an adaptive data partition algorithm, which estimates input data distribution to improve load balance among processors [Shen and Ding, 2004]. The details are out of the scope of this thesis. The following discussions are focused on memory behavior analysis.

2.7.1 Other Research on Reuse Distance

This section discusses the related work in the the measurement and the use of reuse distance.

Performance modeling Reuse distance gives richer information about a program than a cache miss rate does. At least four compiler groups have used reuse distance for different purposes: to study the limit of register reuse [Li et al., 1996] and cache reuse [Huang and Shen, 1996; Ding, 2000; Zhong et al., 2002], and to evaluate the effect of program transformations [Ding, 2000; Beyls and D'Hollander, 2001; Almasi et al., 2002; Zhong et al., 2002]. For cache performance prediction, besides our group's work, Marin and Mellor-Crummey applied distance-based analysis to memory blocks and reported accurate miss-rate prediction across different program inputs and cache sizes[Marin and Mellor-Crummey, 2005, 2004]. Fang et al. [Fang et al., 2005, 2004] examined the reuse pattern per instruction and predicted the miss rate of 90% of instructions with a 97% accuracy. They used the prediction tool to identify *critical* instructions that generate the most cache misses and extended the distance model to memory disambiguation.

Program transformation Beyls and D'Hollander [Beyls and D'Hollander, 2002] is the first to show real performance improvement using the reuse distance information. They used reuse distance profiles to generate cache hints, which tell the hardware whether and which level to place or replace a loaded memory block in cache. Their method improved the performance of SPEC95 CFP benchmarks by an average 7% on an Itanium processor. The next chapter of this thesis presents a technique to sub-divide the whole-program distance pattern in the space of its data. The spatial analysis identifies locality relations among program data. Programs often have a large number of homogeneous data objects such as molecules in a simulated space or nodes in a search tree. Each object has a set of attributes. In Fortran 77 programs, attributes of an object are stored separately in arrays. In C programs, the attributes are stored together in a structure. Neither scheme is sensitive to the access pattern of a program. A better way is to group attributes based on the locality of their access. For arrays, the transformation is array regrouping. For structures, it is structure splitting. We grouped arrays and structure fields that have a similar reuse signature. The new data layout consistently outperformed array and structure layouts given by the programmer, compiler analysis, frequency profiling, and statistical clustering on machines from all major vendors [Zhong et al., 2004].

Memory adaptation A recent trend in memory system design is adaptive caching based on the usage pattern of a running program. Balasubramonian et al. [Balasubramonian et al., 2000a] described a system that can dynamically change the size, associativity, and the number of levels of on-chip cache to improve speed and save energy. To enable phase-based adaptation, our recent work divides the distance pattern in time to identify abrupt reuse-distance changes as phase boundaries. The new technique is shown more effective at identifying long, recurring phases than previous methods based on program code, execution intervals, and manual analysis [Shen et al., 2004b] (see Chapter 4 for details). For FPGA-based systems, So et al. [So et al., 2002] showed that a best design can be found by examining only 0.3% of design space with the help of program information, including the balance between computation and memory transfer as defined by Callahan et al [Callahan et al., 1988b]. So et al. used a compiler to adjust program balance in loop nests and to enable software and hardware co-design. While our analysis cannot change a program to have a particular balance (as techniques such as unroll-and-jam do [Carr and Kennedy, 1994]), it can be used to measure memory balance and support hardware adaptation for programs that are not amenable to loop-nest analysis.

File caching For software managed cache, Jiang and Zhang [Jiang and Zhang, 2002] developed an efficient buffer cache replacement policy, *LIRS*, based on the assumption that the reuse distance of cache blocks is stable over a certain time period. Zhou et al. [Zhou et al., 2001] divided the second-level server cache into multiple buffers dedicated to blocks of different reuse intervals. The common approach is partition cache space into multiple buffers, each holding data of different reuse distances. Both studies showed that reuse distance based management outperforms single LRU cache and other frequency-based schemes. Our work will help in two ways. The first is faster analysis, which reduces the management cost for large buffers (such as server cache), handles larger traces, and provides faster run-time feedbacks. The second is predication, which gives not only the changing pattern but also a quantitative measure of the regularity within and between different types of workloads.

2.7.2 Comparison with Program Analysis Techniques

Data reuse analysis can be performed mainly in three ways: by a compiler, by profiling or by run-time sampling. Compiler analysis can model data reuse behavior for basic blocks and loop nests. An important tool is dependence analysis. Allen and Kennedy's recent book [Allen and Kennedy, 2001] contains a comprehensive discussion on this topic. Various types of array sections can measure data locality in loops and procedures. Such analysis includes linearization for high-dimensional arrays by Burke and Cytron [Burke and Cytron, 1986], linear inequalities for convex sections by

Triolet et al. [Triolet et al., 1986], regular sections by Callahan and Kennedy [Callahan et al., 1988a], and reference list by Li et al. [Li et al., 1990]. Havlak and Kennedy studied the effect of array section analysis on a wide range of programs [Havlak and Kennedy, 1991]. Cascaval extended dependence analysis to estimate the distance of data reuses [Cascaval, 2000]. Other locality analysis includes the unimodular matrix model by Wolfe and Lam [Wolf and Lam, 1991], the memory order by McKinley et al. [McKinley et al., 1996], and a number of recent studies based on more advanced models.

Balasundaram et al. presented a performance estimator for parallel programs [Balasundaram et al., 1991]. A set of kernel routines include primitive computations and common communication patterns are used to train the estimator. While their method trains for different machines, our scheme trains for different data inputs. Compiler analysis is not always accurate for programs with input-dependent control flow and dynamic data indirection. Many types of profiling analysis have been used to study data access patterns. However, most past work is limited to using a single inputs or measuring correlation among a few executions. The focus of this work is to predict changes for new data inputs.

Run-time analysis often uses sampling to reduce the overhead. Ding and Kennedy sampled program data [Ding and Kennedy, 1999a], while Arnold and Ryder sampled program execution [Arnold and Ryder, 2001]. Run-time analysis can identify patterns that are unique to a program input, while training-based prediction cannot. On the other hand, profiling analysis can analyze all accesses to all data.

2.8 Future Directions

Although the models presented in this chapter reduce much prediction overhead, the whole process is still hundreds of times slower than the original program's execution.

The bottleneck is the measurement of reuse distance. It remains an open question to improve the measurement efficiency.

Besides locality, some other kinds of behavior are input-sensitive also, such as running time, concurrency granularity, and energy consumption. The accurate estimation of running time is important for task scheduling. Knowing the running time of each task could help task assignment with better balance; Scheduling small tasks before large ones could improve response time. It is also important for building a cost-benefit model, and thus determining whether a dynamic optimization is worthwhile.

Computers are providing multi-level concurrencies. Forecasting the parallelism level of a program is critical for the intelligent utilization of the extra computing resource. For a phase with few concurrencies, it could be worthwhile to turn off some processors for energy savings and turn them on before a highly parallel phase. That requires the prediction of the concurrency in both short and long term (e.g. in several phases) because of the overhead to shut down and restart a processor.

At the end of the next chapter, we will discuss another future direction, extending the locality analysis, together with reference affinity presented in the next chapter, to a wider range of applications, including parallel programs and object-oriented programs with garbage collections.

2.9 Summary

This chapter presents a novel technique based on regression analysis for locality prediction across program inputs. Through the comparison with 4 approaches, we draw the following conclusions.

1. Regression significantly improves the accuracy of reuse distance prediction, even with only a few training inputs.

2. The multi-model method using logarithmic histograms can save 95% space and computations and still keep the best accuracy in most programs, although it is not as consistent as those methods using log-linear histograms. It is a good choice when efficiency is important.

3. The multi-model method using log-linear scale histograms is the best for small input sizes, where different models tend to overlap each other. It is also efficient because of the small input size.

4. The single-model multi-input method has the highest accuracy, but it cannot accurately model small-size inputs. It is the best choice when one can tolerate a high profiling cost.

Reuse distance prediction allows locality analysis and optimizations to consider program inputs other than profiled ones. The techniques discussed in this work may also help to solve other prediction problems, such as running time and concurrency.

Reuse signature of the whole program reveals the big picture of the locality of the execution. If we focus on a particular object such as an array, we could get per-object reuse signature, which turns out to be critical for improving data locality as shown in the next chapter.

3 Reference Affinity

While the memory of most machines is organized as a hierarchy, program data are laid out in a uniform address space. This chapter defines a model of *reference affinity*, which measures how close a group of data are accessed together in a reference trace. Based on the model, a profiling-based technique, k*-distance analysis*, is briefly described to demonstrate the finding of the hierarchical data affinity. The technique, however, requires detail instrumentation to obtain the reuse signature of a profiling run, the high overhead impeding its adoption in a general compiler.

The main part of this chapter is devoted to a *lightweight affinity model* upon an interprocedural analysis. The technique summarizes the access pattern of an array by a frequency vector and then estimates the affinity of two arrays using their vector distance. Being context sensitive, the analysis tracks the exact array accesses. Using static estimation, the analysis removes all profiling overhead. Implemented in the IBM FORTRAN compiler to regroup arrays in scientific programs, the lightweight analysis achieves similar results as k-distance analysis, and generates data layout consistently outperforming the ones given by the programmer, compiler analysis, frequency profiling and statistical clustering. It suggests the applicability of affinity analysis for general compilers.

3.1 Introduction

All current PCs and workstations use cache blocks of at least 64 bytes, making the utilization an important problem. If only one word is useful in each cache block, a cache miss will not serve as a prefetch for other useful data. Furthermore, the program would waste up to 93% of memory transfer bandwidth and 93% of cache space, causing even more memory access.

To improve cache utilization we need to group related data into the same cache block. The question is how to define the relation. We believe that it should meet three requirements. First, it should be solely based on how data are accessed. For example in an access sequence “$abab..ab$”, a and b are related and should be put in the same cache block, regardless how they are allocated and whether they are linked by pointers. Second, the relation must give a unique partition of data. Consider for example the access sequence “$abab..ab...bcbc..bc$”. Since data a and c are not related, b cannot relate to both of them because it cannot stay in two locations in memory. Finally, the relation should be a scale. Different memory levels have blocks of increasing sizes, from a cache block to a memory page. The grouping of “most related” data into the smallest block should precede the grouping of “next related” data into larger blocks. In summary, the relation should give a unique and hierarchical organization of all program data.

We define such a relation as *reference affinity*, which measures how close a group of data are accessed *together* in an execution. Unlike most other program analysis, we measure the “togetherness” using the *volume distance*, the number of distinct elements accessed between two memory accesses, also called *LRU stack distance*. Notice the volume distance is an extension of *reuse distance*; the latter is a distance between the accesses to the same data but the former could be between the accesses to the different data. As a notion of locality, volume distance is bounded, even for long-running programs. The long volume distance often reveals long-range data access patterns that may otherwise hide behind complex control flows, indirect data access, or variations in coding and data allocation. We prove that the new definition gives a unique partition of program data for each distance k. When we decrease the value of k, the reference affinity gives a hierarchical decomposition and finds data sub-groups with closer affinity, much in the same way we sharpen the focus by reducing the radius of a circle.

Chapter 2 shows that the *reuse signature*, the histogram of reuse distance, has a consistent pattern across all data inputs even for complex programs or regular programs after complex compiler optimizations. This suggests that we can analyze the reference affinity of the whole program by looking at its reuse signatures from training runs.

We present k-distance analysis, which simplifies the requirements of reference affinity into a set of necessary conditions about reuse signatures. The simplified conditions can then be checked efficiently for large, complex programs. The parameter k has an intuitive meaning—elements in the same group are almost always used within a distance of k data elements. The analysis handles sequential programs with arbitrarily complex control flows, indirect data access, and dynamic memory allocation. The analysis uses multiple training runs to take into account the variation caused by program inputs.

Reuse-distance profiling, however, carries a high overhead. The slowdown is at least 10 to 100 times. No production compiler is shipped with such a costly technique. No one would before a careful examination whether such a high cost is justified.

To solve that problem, we present a lightweight technique, *frequency-based affinity analysis*. It uses a frequency-based model to group arrays even if they are not always accessed together. It uses interprocedural program analysis to measure the access frequency in the presence of array parameters and aliases. To collect the frequency within a loop or a function, we study two methods. The first is symbolic analysis by a compiler. The second is lightweight profiling.

The rest of the chapter is organized as follows. Section 3.2 briefly presents the distance-based affinity model and the analysis technique. (Chapter 5 and 6 of Zhong's thesis have the details [Zhong, 2005].) Section 3.3 describes the frequency-based affinity mode and the analysis method. Section 3.4 introduces two data reorganization techniques. Secion 3.5 demonstrates the effectiveness of the affinity analysis in helping data reorganization to improve locality and thus speed up programs. The chapter concludes with the related work and future directions.

3.2 Distance-based Affinity Analysis

3.2.1 Distance-Based Reference Affinity Model

The affinity model is based on several concepts. An *address trace* or *reference string* is a sequence of accesses to a set of data elements. We use letters such as x, y, z to represent data elements, subscripted symbols such as a_x, a'_x to represent accesses to a particular data element x, and the array index $T[a_x]$ to represent the logical time of the access a_x on a trace T.

Volume distance is the number of distinct elements accessed between two memory accesses, also called *LRU stack distance*.

Based on the volume distance, we define a *linked path* in a trace. It is parameterized by a distance bound k. There is a linked path from a_x to a_y ($x \neq y$) if and only if there exist t accesses, a_{x_1}, a_{x_2}, $\ldots$, a_{x_t}, such that (1) $dis(a_x, a_{x_1}) \leq k \wedge$ $dis(a_{x_1}, a_{x_2}) \leq k \wedge \ldots \wedge dis(a_{x_t}, a_y) \leq k$ and (2) $x_1, x_2, \ldots, x_t$, x and y are different data elements.

We now present the formal definition of the reference affinity.

Definition 1. Strict Reference Affinity. *Given an address trace, a set G of data elements is a strict affinity group (i.e. they have the reference affinity) with the link length k if and only if*

1. *for any $x \in G$, all its accesses a_x must have a linked path from a_x to some a_y for each other member $y \in G$, that is, there exist different elements $x_1, x_2, \ldots, x_t \in G$ such that $dis(a_x, a_{x_1}) \leq k \wedge dis(a_{x_1}, a_{x_2}) \leq k \wedge \ldots \wedge dis(a_{x_t}, a_y) \leq k$.*

2. *adding any other element to G will make Condition (1) impossible to hold.*

Three properties have been proved about strict reference affinity groups [Zhong, 2005]:

- *Unique partition*: for a given link length, the data partition is consistent, i.e. each data element belongs to one and only one affinity group.
- *Hierarchical structure*: an affinity group with a shorter link length is a subset of an affinity group with a greater link length.
- *Bounded distance*: when one element is accessed, all other elements will be accessed within a bounded volume distance.

The strict affinity requires that the members of an affinity group be always accessed together. On most machines, it is still profitable to group data that are almost always accessed together because the side effect would not outweigh the benefit.

3.2.2 k-Distance Affinity Analysis

Distance-based affinity analysis is a profiling-based technique to measure the "almost strict" reference affinity in complex programs. It detects affinity groups according to the similarity of the reuse distance histograms of the sets of data in the program. That's a necessary but not sufficient condition.

Let a reuse histogram have B bins after removing short-distance bins. Let X and Y be the two sets of data, and Avg_i^X and Avg_i^Y be the average reuse distance of the two data sets in the ith bin.

$$d = \sum_{i=1}^{B} |Avg_i^X - Avg_i^Y| \leq k \cdot B \tag{3.1}$$

The equation ensures that the average reuse distance per bin differs by no more than k. The left-hand side of the inequality is the difference between X and Y known as the Manhattan distance of the two vectors.

A reuse distance does not include the exact time of the data access. It is possible that two elements are accessed in the same reuse distance, but one in the first half of the execution, and the other in the second half. An improvement is to divide the execution trace into sub-parts and check the condition for each part.

The maximal difference between any two members of a g-element affinity group is no more than $2gk$. For each data set X, we find all other sets whose average distance differs no more than bk and let b range from 1 to $2g$. The solution is the largest b such that exactly $b - 1$ data sets satisfy the condition. The process must terminate with the correct result.

In practice, we use a stricter condition to build a group incrementally. Initially each data set is a group. Then we traverse the groups and merge two groups if a member in one group and another member in the other group satisfy Equation 3.1. The process terminates if no more groups can be merged. The reference affinity forms a hierarchy for different k values. Interested readers please refer to Zhong's thesis [Zhong, 2005] for details.

3.3 Lightweight Frequency-Based Affinity Analysis

The distance-based analysis requires the monitoring of every memory access to collect reuse distance histograms, the overhead posing a big challenge for the use in a product compiler. This section presents a lightweight affinity analysis based on interprocedure-estimation of the access frequencies of data. The analysis shows similar effectiveness as distance-based analysis for floating-point FORTRAN programs.

First of all, the affinity model becomes different from the one used in the distance-based analysis. In the new model, arrays are nodes and affinities are edge weights in the affinity graph, and the affinity groups are obtained through linear-time graph partitioning.

3.3.1 Frequency-Based Affinity Model

A program is modeled as a set of code units, in particular, loops. Suppose there are K code units. Let f_i represent the total occurrences of the ith unit in the program execution. We use $r_i(A)$ to represent the number of references to data A in an execution of the ith unit. The frequency vector of data A is defined as follows:

$$V(A) = (v_1, v_2, \ldots, v_K)$$

where

$$v_i = \begin{cases} 0 & \text{if } r_i(A) = 0; \\ f_i & \text{if } r_i(A) > 0. \end{cases}$$

A code unit i may have branches inside and may call other functions. We conservatively assume that a branch goes both directions when collecting the data access. We use interprocedural analysis to find the side effects of function calls.

To save space, we can use a bit vector to replace the access vector of each data and use a separate vector to record the frequency of code units.

The affinity between two data is the Manhattan distance between their access-frequency vectors, as shown below. It is a number between zero and one. Zero means that two data are never used together, while one means that both are accessed whenever one is. The formula is as follows with δ equal to 0.0001 to avoid zero divider.

$$affinity(A, B) = 1 - \frac{\sum_{i=1}^{K} \mid (v_i(A) - v_i(B)) \mid}{\delta + \sum_{i=1}^{K} (v_i(A) + v_i(B))}$$

We construct an affinity graph. Each node represents a data, and the weight of an edge between two nodes is the calculated affinity between them. There are additional constraints. To be regrouped, two data must be compatible. As arrays for example, they should have the same number of elements and they should be accessed in the same order [Ding and Kennedy, 2004]. The data access order is not always possible to analyze at compile time. However, when the information is available to show that two arrays are not accessed in the same order in a code unit, the weight of their affinity edge will be reset to zero. The same is true if two arrays differ in size.

Graph partitioning gives reference affinity groups; the closeness in a group is determined by the partition threshold values. The next section presents the technical detail on program units, frequency estimation, and graph partitioning.

3.3.2 Lightweight Affinity Analysis Techniques

3.3.2.1 Unit of Program Analysis

For scientific programs, most data accesses happen in loops. We use a loop as a hot code unit for frequency counting for three reasons: coverage, independence, and efficiency.

- *Coverage:* A loop often accesses an entire array or most of an array. In that case, branches and function calls outside the loop have no effect on whether two arrays are accessed together or not.
- *Independence:* McKinley and Temam reported that most cache misses in SPEC95 FP programs were due to cross-loop reuses [McKinley and Temam, 1999]. We expect the same for our test programs and ignore the cache reuse across two loops. Therefore, the temporal order in which loops are executed has no effect on the affinity relation. Without the independence, when two arrays appear in different code units, their affinity may depend on the temporal relations across units. The independence property simplifies the affinity analysis by allowing it to compose the final result from analyzing individual code units.
- *Efficiency:* The total number of loops determines the size of the access-frequency vector. In a context-sensitive analysis, a unit becomes multiple elements in the

access-frequency vector, one for each distinct calling context. The number of loops is small enough to enable full context-sensitive analysis, as described in Section 3.3.2.4. In our experiment, the maximum is 351 for benchmark Galgel.

In comparison, other types of code units are not as good for array regrouping. For example, a basic block has too little data access to be independent from other basic blocks. Basic blocks may be too numerous for compiler analysis or lightweight profiling to be affordable. A small procedure lacks independence in data access. A large procedure has less coverage because it often has a more complex control flow than a loop does. Other possible code units are super-blocks and regions, but none satisfies the three requirements as well as loops do. Loops have good independence, so the temporal order of loops has little impact on the affinity result. The number of loops is not overly large in most programs. Branches inside loops hurt the coverage. However, very few branches exist in loops in scientific programs, especially in the innermost loop.

3.3.2.2 Static Estimate of the Execution Frequency

Many past studies have developed compiler-based estimate of the execution frequency (e.g., [Sarkar, 1989; Wagner et al., 1994]). The main difficulties are to estimate the value of a variable, to predict the outcome of a branch, and to cumulate the result for every statement in a program. We use standard constant propagation and symbolic analysis to find constants and relations between symbolic variables.

We classify loops into three categories. The bounds of the first group are known constants. The second group of loops have symbolic bounds that depend on the input, e.g. the size of the grid in a program simulating a three-dimensional space. The number of iterations can be represented by an expression of a mix of constants and symbolic values. We need to convert a symbolic expression into a number because the later affinity analysis is based on numerical values. The exact iteration count is impossible to obtain. To distinguish between high-trip count loops from low-trip count loops, we assume that a symbolic value is reasonably large (100) since most low-trip count loops have a constant bound. This strategy works well in our experiments.

The third category includes many while-loops, where the exit condition is calculated in each iteration. Many while-loops are small and do not access arrays, so they are ignored in our analysis. In other small while-loops, we take the size of the largest array referenced in the loop as the number of iterations. If the size of all arrays is unknown, we simply assign a constant 100 as the iteration count.

The array regrouping is not very sensitive to the accuracy of loop iteration estimations. If two arrays are always accessed together, they would be regarded as arrays with perfect affinity regardless how inaccurate the iteration estimations are. Even for arrays without perfect affinity, the high regrouping threshold provides good tolerance of estimation errors as discussed in Section 3.5.

The frequency of the innermost loop is the product of its iteration count, the number of iterations in all enclosing loops in the same procedure, and the estimated frequency of the procedure invocation. The execution frequency of loops and subroutines is estimated using the same interprocedural analysis method described in Section 3.3.2.4. It roughly corresponds to in-lining all procedural calls.

For branches, we assume that both paths are taken except when one branch leads to the termination of a program, i.e. the stop statement. In that case, we assume that the program does not follow the exist branch. This scheme may overestimate the affinity relation. Consider a loop whose body is a statement with two branches α and β. Suppose arrays a is accessed in the α branch and b in the β branch. In an execution, if the two branches are taken in alternative loop iterations, then the affinity relation is accurate, that is, the two arrays are used together. However, if α is taken in the first half iterations and β in the second half (or vice versa), then the two arrays are not used together. The static result is an overestimate.

3.3.2.3 Profiling-Based Frequency Analysis

By instrumenting a program, the exact number of iterations becomes known for the particular input. To consider the effect of the entire control flow, we count the frequency of execution of all basic blocks. Simple counting would insert a counter and an increment instruction for each basic block. In this work, we use the existing implementation in the IBM compiler [Silvera et al., unpublished], which implements more efficient counting by calculating from the frequency of neighboring blocks, considering a flow path, and lifting the counter outside a loop. Its overhead is less than 100% for all programs we tested. The execution frequency for an innermost loop is the frequency of the loop header block. When a loop contains branches, the analysis is an overestimate for reasons described in Section 3.3.2.2.

3.3.2.4 Context-Sensitive Lightweight Affinity Analysis

Aliases in FORTRAN programs are caused by parameter passing and storage association. We consider only the first cause. We use an interprocedural analysis based on the invocation graph, as described by Emami et al [Emami et al., 1994]. Given a program, the invocation graph is built by a depth-first traversal of the call structure starting from the program entry. Recursive call sequences are truncated when the same procedure is called again. In the absence of recursion, the invocation graph enumerates all calling contexts for an invocation of a procedure. A special back edge is added in the case of a recursive call, and the calling context can be approximated.

The affinity analysis proceeds in two steps. The first step takes one procedure at a time, treats the parameter arrays as independent arrays, identifies loops inside the procedure, and the access vector for each array.

The affinity analysis proceeds in two steps. The first step takes one procedure at a time, treats the parameter arrays as independent arrays, identifies loops inside the

procedure, and the access vector for each array. The procedure is given by *BuildStaticAFVList* in Figure 3.1.

The second step traverses the invocation graph from the bottom up. At each call site, the affinity results of the callee are mapped up to the caller based on the parameter bindings, as given by procedures *BuildDynamicAFVList*, *UpdateAFVList*, and *UpdateDyn* in Figure 3.1. As an implementation, the lists from all procedures are merged in one vector, and individual lists are extracted when needed, as in *UpdateDyn*. The parameter binding for a recursive call is not always precise. But a fixed point can be obtained in linear time using an algorithm proposed by Cooper and Kennedy (Section 11.2.3 of [Allen and Kennedy, 2001]).

Because of the context sensitivity, a loop contributes multiple elements to the access-frequency vector, one for every calling context. In the worst case, the invocation graph is quadratic to the number of call sites. However, Emami et al. reported on average 1.45 invocation nodes per call site for a set of C programs [Emami et al., 1994]. We saw a similar small ratio in FORTRAN programs.

The calculation of the access-frequency vector uses the execution frequency of each loop, as in procedure *UpdateDyn*. In the case of static analysis, the frequency of each invocation node is determined by all the loops in its calling context, not including the back edges added for recursive calls. The frequency information is calculated from the top down. Indeed, in our implementation, the static frequency is calculated *at the same time* as the invocation graph is constructed.

The frequency from the lightweight profiling can be directly used if the profiling is context sensitive. Otherwise, the average is calculated for the number of loop executions within each function invocation. The average frequency is an approximation.

The last major problem in interprocedural array regrouping is the consistency of data layout for parameter arrays. Take, for example, a procedure that has two formal parameter arrays. It is called from two call sites; each passes a different pair of actual parameter arrays. Suppose that one pair has reference affinity but the other does not.

To allow array regrouping, we will need two different layouts for the formal parameter arrays. One possible solution is procedural cloning, but this leads to code expansion, which can be impractical in the worst case. In this work, we use a conservative solution. The analysis detects conflicts in parameter layouts and disables array regrouping to resolve a conflict. In the example just mentioned, any pair of arrays that can be passed into the procedure are not regrouped. In other words, array regrouping guarantees no need of code replication in the program.

The invocation graph excludes pointer-based control flow and some use of dynamically loaded libraries. The former does not exist in FORTRAN programs and the latter is a limitation of static analysis.

3.3.2.5 Implementation

As a short summary, the frequency-based analysis includes the following steps:

1. Building control flow graph and invocation graph with data flow analysis
2. Estimating the execution frequency through either static analysis or profiling
3. Building array access-frequency vectors using interprocedural analysis, as shown in Figure 3.1
4. Calculating the affinity between each array pair and constructing the affinity graph
5. Partitioning the graph to find affinity groups in linear time

This work has been implemented in the IBM TPO (Toronto Portable Optimizer), which is the core optimization component in IBM C/C++ and FORTRAN compilers. It implements both compile-time and link-time methods for intra- and interprocedural optimizations. It also implements profiling feedback optimizations. We now describe the structure of TPO and the implementation of the reference affinity analysis.

TPO uses a common graph structure based on Single Static Assignment form (SSA) [Allen and Kennedy, 2001] to represent the control and data flow within a procedure. Global value numbering and aggressive copy propagation are used to perform symbolic analysis and expression simplifications. It performs pointer analysis and constant propagation using the same basic algorithm from Wegman and Zadeck [Wegman and Zadeck, 1985], which is well suited for using SSA form of data flow. For loop nests, TPO performs data dependence analysis and loop transformations after data flow optimizations. We use symbolic analysis to identify the bounds of arrays and estimate the execution frequency of loops. We use dependence analysis to identify regular access patterns to arrays.

During the link step, TPO is invoked to re-optimize the program. Having access to the intermediate code for all the procedures in the program, TPO can significantly improve the precision of the data aliasing and function aliasing information. Interprocedural mod-use information is computed at various stages during the link step.

The reference affinity analysis is implemented at the link step. A software engineering problem is whether to insert it before or after loop transformations. Currently the analysis happens first, so arrays can be transformed at the same compilation pass as loops are. As shown later, early analysis does not lead to slower performance in any of the test programs. We are looking at implementation options that may allow a later analysis when the loop access order is fully determined.

We have implemented the analysis that collects the static access-frequency vector and the analysis that measures per-basic-block execution frequency through profiling. We have implemented a compiler flag that triggers either static or profiling-based affinity analysis. The invocation graph is part of the TPO data structure. We are in the process of completing the analysis that includes the complete context sensitivity. The current access-frequency vector takes the union of all contexts. We have implemented the reference affinity graph and the linear-time partitioning. The array transformations are semi-automated as the implementation needs time to fully bond inside the compiler.

The link step of TPO performs two passes. The first is a forward pass to accumulate and propagate constant and pointer information within the entire program. Reference affinity analysis is part of the global reference analysis used for remapping global data structures. It can clone a procedure [Allen and Kennedy, 2001] when needed, although we do not use cloning for array regrouping. The second pass traverses the invocation graph backward to perform various loop transformations. Interprocedural code motion is also performed during the backward pass. This transformation will move upward from a procedure to all of its call points. Data remapping transformations, including array regrouping when fully implemented, are performed just before the backward pass to finalize the data layout. Loop transformations are performed during the backward pass to take full advantage of the interprocedural information. Interprocedural mod-use information is recomputed again in order to provide more accurate information to the back-end code generator.

3.4 Affinity-Oriented Data Reorganization

Reorganize data according to their affinity could improve program locality and thus effective memory bandwidth. We experimented with two reorganization techniques: structure splitting and array regrouping. This section describes the two techniques followed by the experiment results.

3.4.1 Structure Splitting

The elements of a structure may have different affinity among them. Structure splitting is to split a structure into multiple ones, each including the elements with good affinity. Figure 3.2 illustrates the splitting of one structure N into two when element val and $left$ are always accessed together. There are many issues to implement auto-

Data Structure

$staticAFVList$: the list of local frequency vectors, one per array per subroutine
$dynAFVList$: the list of global frequency vectors, one per array
$loopFreq$: the local estimate of the execution frequency of a loop
$IGNode$: a node in the invocation graph, with the following attributes
  $freq$: the estimated frequency of the node
  $staticStartId$: the position of the subroutine's first loop in staticAFVList vectors
  $dynStartId$: the position of the subroutine's first loop in dynAFVList vectors
$groupList$: the list of affinity groups

Algorithm

1) building control flow graph and invocation graph with data flow analysis
2) estimating the execution frequency (Section 3.3.2.2 and 3.3.2.3)
3) building array access-frequency vectors using interprocedural analysis (Section 3.3.2.4)
4) calculating the affinity and constructing the affinity graph (Section 3.3.1)
5) linear-time graph partitioning to find affinity groups (Section 3.3.1)

Procedure BuildAFVList()

```
// build access frequency vectors
   BuildStaticAFVList ();
   BuildDynamicAFVList ();
End
```

Procedure BuildStaticAFVList()

```
// local access frequency vectors
  id = 0;
  For each procedure proc
    For each inner-most loop l in proc
      refSet = GetArrayRefSet(l);
      If (refSet == NULL)
        Continue;
      End
      id ++;
      For each member a in refSet
        staticAFVList[a][id]
                   =loopFreq(l);
      End
    End
  End
End
```

Procedure BuildDynamicAFVList()

```
// global access frequency vectors
  For each leaf node n in
              the invocation graph
    UpdateAFVList(n);
  End
End
```

Procedure UpdateAFVList(IGNode n)

```
  For each array a in n.refSet
    UpdateDyn(a,n);
  End
  par = n.Parent();
  if (par == NULL) return;
  For each array virtual parameter p
    q = GetRealParameter(p);
    UpdateDyn(q,n);
  End
  n.visited = true;
  If (IsAllChildrenUpdated(par))
    UpdateAFVList(par);
  End
End
```

Procedure UpdateDyn(array a, IGNode n)

```
  s1=n.staticStartId;
  s2=n.dynStartId;
  i=0;
  While (i< n.loopNum)
    dynAFVList[a][s2+i]
       += staticAFVList[a][s1+i]*n.freq;
    i++;
  End
End
```

Procedure GraphPartition()

```
// partition into affinity groups
  For each edge e in the affinity graph g
    If (edge.affinity> Threshold)
      g.merge(edge);
    End
  End
  groupList = g.GetNodeSets();
End
```

Figure 3.1: Interprocedural reference affinity analysis

matic structure splitting such as addressing problem; details are in Chapter 6 of Zhong's thesis [Zhong, 2005].

```
struct N {
    int val;
    struct N* left;
    struct N* right;
};
```

(a) before splitting

```
struct N_fragm0 {
    int val;
    unsigned int left;
};
struct N_fragm1 {
    unsigned int right;
};
```

(b) after splitting

Figure 3.2: Structure splitting example in C

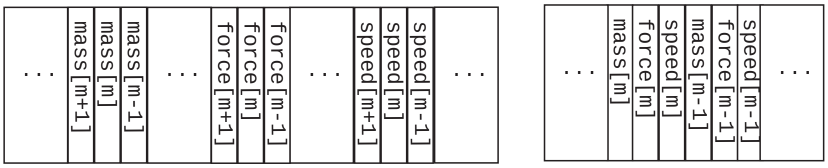
3.4.2 Array Regrouping

Figure 3.3 shows an example of array regrouping. Part (a) shows a program that uses four attributes of N molecules in two loops. One attribute, “$position$", is used in both the compute loop and the visualization loop, but the other three are used only in the compute loop. Part (b) shows the initial data layout, where each attribute is stored in a separate array. In the compute loop, the four attributes of a molecule are used together, but they are stored far apart in memory. On today's high-end machines from IBM, Microsystems, and companies using Intel Itanium and AMD processors, the largest cache in the hierarchy is composed of blocks of no smaller than 64 bytes. In the worst case, only one 4-byte attribute is useful in each cache block, 94% of cache space would be occupied by useless data, and only 6% of cache is available for data reuse. A similar issue exists for memory pages, except that the utilization problem can be much worse.

Array regrouping improves spatial locality by grouping three of the four attributes together in memory, as shown in part (c) of Figure 3.3.After regrouping, a cache block should have at least three useful attributes. One may suggest grouping all four attributes. However, three of the attributes are not used in the visualization loop, and

```
% attributes for molecules
position[N], speed[N], mass[N]
force[N]
... ...
compute_loop
  position[m]=f(position[m], speed[m],
               force[m], mass[m])
... ...
visualization_loop
  display(position[k])
... ...
```

(a) a program that uses four arrays



(b) initially all arrays are separately stored   **(c) three arrays are grouped together**

Figure 3.3: An example of array regrouping. Data with reference affinity are placed together to improve cache utilization

therefore grouping them with “position” hurts cache-block utilization. However, if the loop is infrequently executed or it touches only a few molecules, then we may still benefit from grouping all four attributes.

Array regrouping has many other benefits. First, it reduces the interference among cache blocks because fewer cache blocks are accessed. By combining multiple arrays, array regrouping reduces the page-table working set and consequently the number of Translation Lookaside Buffer (TLB) misses in a large program. It also reduces the register pressure because fewer registers are needed to store array base addresses. It may improve energy efficiency by allowing more memory pages to enter a sleeping model. For the above reasons, array regrouping is beneficial even for arrays that are contiguously accessed.

These benefits have been verified in our previous study [Ding and Kennedy, 2004]. Finally, on shared-memory parallel machines, better cache-block utilization means slower amortized communication latency and better bandwidth utilization.

Array regrouping is mostly orthogonal to traditional loop-nest transformations and single-array transformations. The latter two try to effect contiguous access within a single array. Array regrouping complements them by exploiting cross-array spatial locality, even when per-array data access is contiguous. As a data transformation, it is applicable to irregular programs where the dependence information is lacking. In the example in Figure 3.3, the correctness of the transformation does not depend on knowing the value of index variables m and k. While array regrouping has a good potential for complex programs, it has not been implemented in any production compiler because the current techniques are not up to the task.

Ding and Kennedy gave the first compiler technique for array regrouping [Ding and Kennedy, 2004]. They defined the concept *reference affinity*. A group of arrays have reference affinity if they are *always* accessed together in a program. Their technique is conservative and groups arrays only when they are always accessed together. We call this scheme *conservative affinity analysis*. Conservative analysis is too restrictive in real-size applications, where many arrays are only sporadically accessed.

3.5 Evaluation

In this section, we focus on array regrouping in FORTRAN programs. Zhong's thesis presents the results of distance-based affinity analysis for structure splitting on a wider range of programs [Zhong, 2005].

Array regrouping experiments are conducted with 11 benchmarks on two machines as shown in table 3.1. Table 3.2 gives the source and a description of the test programs. Eight are from SPEC CFP2000. The other three are programs utilized in the evaluation of distance-based affinity analysis [Zhong et al., 2004]. Most of them are scientific simulations for quantum physics, meteorology, fluid and molecular dynamics. Two are on image processing and number theory. They have 4 to 92 arrays.

Table 3.1: Machine architectures for affinity study

Machine Type	IBM p690 Turbo+	Intel PC
Processor	Power4+ 1.7GHz	Pentium 4 2.8GHz
L1 data cache	32KB, 2-way, 128B cache line	8KB, 64B cache line
L2 data cache	1.5MB, 4-way	512KB, 8-way

Table 3.2: Test programs in affinity experiments

Benchmark	**Source**	**Description**	**Arrays**
Applu	Spec2K	Physics / Quantum Chromodynamics	38
Apsi	Spec2K	Meteorology: Pollutant Distribution	92
Facerec	Spec2K	Image Processing: Face Recognition	44
Galgel	Spec2K	Computational Fluid Dynamics	75
Lucas	Spec2K	Number Theory / Primality Testing	14
Mgrid	Spec2K	Multi-grid Solver:3D Potential Field	12
Swim2K	Spec2K	Shallow Water Modeling	14
Wupwise	Spec2K	Physics / Quantum Chromodynamics	20
Swim95	Zhong+	Shallow Water Modeling	14
Tomcatv	Zhong+	Vectorized Mesh Generation	9
MolDyn	Zhong+	Molecular Dynmaics Simulation	4

3.5.1 Affinity Groups

Table 3.3 shows the affinity groups identified by interprocedural reference affinity analysis using static estimates. The program that has most non-trivial affinity groups is $Galgel$. It has eight affinity groups, including 24 out of 75 arrays in the program. Four programs—$Apsi$, $Lucas$, $Wupwise$, and $MolDyn$—do not have affinity groups with more than one array. $Apsi$ uses only one major array, although different parts as many different arrays in over 90 subroutines. It is possible to split the main array into many smaller pieces. It remains our future work. $Lucas$, $Wupwise$, and $MolDyn$ have multiple arrays but no two have strong reference affinity. The affinity groups in $Facerec$ and $Mgrid$ contain only small arrays. The other three SPEC CPU2000 programs, $Applu$, $Galgel$, and $Swim2K$, have reference affinity among large arrays.

Table 3.3: Affinity groups

Benchmark	**Affinity groups**
Applu	(imax,jmax,kmax) (idmax,jdmax,kdmax) (phi1,phi2) (a,b,c) (ldx,ldy,ldz) (udx,udy,udz)
Apsi	<none>
Facerec	(coordx,coordy)
Galgel	(g1,g2,g3,g4) (f1,f2,f3,f4) (vyy,vyy2,vxy,vxy2) (vxxx,vyxx) (vyyy,vxxy,vxyy,vyxy) (v1,v2) (wxtx,wytx) (wypy,wxpy)
Lucas	<none>
Mgrid	(j1,j2,j3)
Swim2K	(unew,vnew,pnew) (u,v) (uold,vold,pold) (cu,cv,z,h)
Wupwise	<none>
Swim95	(unew,vnew,pnew) (u,v) (uold,vold,pold) (cu,cv,z,h)
	compare to [Zhong et al., 2004]: (unew,vnew,pnew) (u,v) (uold,pold) (vold) (cu,cv,z,h)
Tomcatv	(x,y) (rxm,rym) (rx,ry)
	compare to [Zhong et al., 2004]: (x,y) (rxm,rym) (rx,ry)
MolDyn	< none >
	compare to [Zhong et al., 2004]: <none>

3.5.2 Comparison with Distance-Based Affinity Analysis

$Swim95$, $Tomcatv$, and $MolDyn$ are three FORTRAN programs also tested using distance-based analysis. The profiling time is in hours for a program.

The bottom six rows of Table 3.3 compare the affinity groups from distance-based analysis. Frequency-based analysis gives the same result for $Tomcatv$ and $MolDyn$ without any profiling. The results for $Swim95$ differ in one of the four non-trivial groups. Table 3.4 shows the performance difference between the two layouts on IBM and Intel machines. At “-O3”, the compiler analysis gives better improvement than distance-based profiling. The two layouts have the same performance at “-O5”, the highest optimization level. Without any profiling, the frequency-based affinity analysis is as effective as distance-based affinity analysis.

Table 3.4: Comparison of compiler and K-distance analysis on $Swim95$

			Static	K-distance
Groups			unew,vnew,pnew u,v uold,vold,pold cu,cv,z,h	unew,vnew,pnew u,v uold,pold cu,cv,z,h
IBM	-03	time	17.1s	17.6s
		speedup	96%	90%
	-05	time	15.2s	15.3s
		speedup	91%	91%
Intel	-03	time	41.2s	42.3s
		speedup	48%	44%
	-05	time	34.9s	34.9s
		speedup	42%	42%

3.5.3 Comparison with Lightweight Profiling

The lightweight profiling gives the execution frequency of loop bodies and call sites. These numbers are used to calculate data-access vectors. The resulting affinity groups are the same compared to the pure compiler analysis. Therefore, code profiling does not improve the regrouping results of the analysis. One exception, however, is when a program is transformed significantly by the compiler. The profiling results reflect the behavior of the optimized program, while our compiler analysis measures the behavior of the source program. Among all test programs, $Swim2K$ and $Swim95$ are the only ones in which binary-level profiling of the optimized program yields different affinity groups than compiler analysis.

3.5.4 Performance Improvement from Array Regrouping

Table 3.5 and Table 3.6 show the speedup on IBM and Intel machines, respectively. We include only programs where array regrouping is applied. Each program is compiled with both “-O3” and “-O5” optimization flags. At “-O5” on IBM machines, array regrouping obtained more than 10% improvement on $Swim2K$, $Swim95$, and

$Tomcatv$, 2-3% on $Applu$ and $Facerec$, and marginal improvement on $Galgel$ and $Mgrid$. The improvement is significantly higher at “-O3”, at least 5% for all but $Mgrid$. The difference comes from the loop transformations, which makes array access more contiguous at “-O5” and reduces the benefit of array regrouping. The small improvements for $Facerec$ and $Mgrid$ are expected because only small arrays show reference affinity.

Our Intel machines did not have a good FORTRAN 90 compiler, so Table 3.6 shows results for only FORTRAN 77 programs. At “-O5”, array regrouping gives similar improvement for $Swim2K$ and $Swim95$. It is a contrast to the different improvement on IBM, suggesting that the GNU compiler is not as highly tuned for SPEC CPU2000 programs as the IBM compiler is. $Applu$ runs slower after array regrouping on the Intel machine. The regrouped version also runs 16% slower at “-O5” than “-O3”. We are investigating the reason for this anomaly.

Table 3.5: Execution time (sec.) on IBM Power4+

Benchmark	**-03 Optimization**		**-05 Optimization**	
	Original	**Regrouped (speedup)**	**Original**	**Regrouped (speedup)**
Applu	176.4	136.3 (29.4%)	161.2	157.9 (2.1%)
Facerec	148.6	141.3 (5.2%)	94.2	92.2 (2.2%)
Galgel	123.3	111.4 (10.7%)	83.2	82.6 (0.7%)
Mgrid	231.4	230.1 (0.6%)	103.9	103.0 (0.9%)
Swim2K	236.8	153.7 (54.1%)	125.2	110.1 (13.7%)
Swim95	33.6	17.1 (96.5%)	29.0	15.2 (90.8%)
Tomcatv	17.3	15.4 (12.3%)	16.8	15.1 (11.3%)

3.5.5 Choice of Affinity Threshold

In the experiment, the affinity threshold is set at 0.95, meaning that for two arrays to be grouped, the normalized Manhattan distance between the access-frequency vectors is at most 0.05. To evaluate how sensitive the analysis is to this threshold, we apply

Table 3.6: Execution time (sec.) on Intel Pentium IV

Benchmark	**-03 Optimization**		**-05 Optimization**	
	Original	**Regrouped (speedup)**	**Original**	**Regrouped (speedup)**
Applu	427.4	444.0 (-3.7%)	429.4	444.6 (-3.4%)
Facerec	-	-	-	-
Galgel	-	-	-	-
Mgrid	461.7	460.6 (0.2%)	368.9	368.1 (0.2%)
Swim2K	545.1	315.7 (72.7%)	408.8	259.4 (57.6%)
Swim95	61.1	41.2 (48.3%)	49.7	34.9 (42.4%)
Tomcatv	48.8	44.5 (9.7%)	40.9	37.8 (8.2%)

Table 3.7: Gap between the top two clusters of affinity values

Benchmark	**Cluster-I lower boundary**	**Cluster-II upper boundary**
Applu	0.998	0.667
Apsi	1	0.868
Facerec	0.997	0.8
Galgel	1	0.8
Lucas	1	0.667
Mgrid	1	0.667
Swim	0.995	0.799
Swim95	0.995	0.799
Tomcatv	1	0.798
Wupwise	1	0.8

X-means clustering to divide the affinity values into groups. Table 3.7 shows the lower boundary of the largest cluster and the upper boundary of the second largest cluster. All programs show a sizeable gap between the two clusters, 0.13 for Apsi and more than 0.2 for all other programs. Any threshold between 0.87 and 0.99 would yield the same affinity groups. Therefore, the analysis is quite insensitive to the choice of the threshold.

3.6 Related Work

3.6.1 Affinity Analysis

Early compiler analysis identifies groups of data that are used together in loop nests. Thabit used the concept of pair-wise affinity he called reference proximity [Thabit, 1981]. Wolf and Lam [Wolf and Lam, 1991] and McKinley et al. [McKinley et al., 1996] used reference groups. But none of them considered the interprocedure complexity.

Program profiling has long been used to measure the frequency of data access [Knuth, 1971]. Seidl and Zorn grouped frequently accessed objects to improve virtual memory performance [Seidl and Zorn, 1998]. Using pair-wise affinity, Calder et al. [Calder et al., 1998] and Chilimbi et al. [Chilimbi et al., 1999b] developed algorithms for hierarchical data placement in dynamic memory allocation. The locality model of Calder et al. was an extension of the temporal relation graph of Gloy and Smith, who considered reuse distance in estimating the affinity relation [Gloy and Smith, 1999]. The pair-wise affinity forms a complete graph where each datum is a node and the pair-wise frequency is the edge weight. However, the reference affinity is not transitive in a (pair-wise) graph. Consider the access sequence $abab..ab$ ... $bcbc..bc$: the pair-wise affinity exists for a and b, for b and c, but not for a and c. Hence the pair-wise affinity is indeed pair wise and cannot guarantee the affinity relation for data groups with more than two elements. Furthermore, the criterion for two data “accessed together" is based on preselected “cut-off” radii. In comparison, k-distance analysis defines affinity in data groups and measures the “togetherness” with a scale—the data volume between accesses.

3.6.2 Data Transformations

Thabit showed that the optimal data placement using the pair-wise affinity is NP-hard [Thabit, 1981]. Kennedy and Kremer gave a general model that considered, among others, run-time data transformation. They also showed that the problem is NP-hard [Kennedy and Kremer, 1998]. Ding and Kennedy used the results of Thabit and of Kennedy and Kremer to prove the complexity of the partial and dynamic reference affinity [Ding and Kennedy, 1999b]. To reduce false sharing in multi-treaded programs, Anderson et al. [Anderson et al., 1995] and Eggers and Jeremiassen [Jeremiassen and Eggers, 1995] grouped data accessed by the same thread. Anderson et al. optimized a program for computation as well as data locality, but they did not combine different arrays. Eggers and Jeremiassen combined multiple arrays for thread locality, but their scheme may hurt cache locality if not all thread data are used at the same time.

For improving the cache performance, Ding and Kennedy grouped arrays that are always used together in a program [Ding and Kennedy, 1999b]. They gave the optimal array layout for strict affinity. They later grouped arrays at multiple granularity [Ding and Kennedy, 2004]. An earlier version of the distance-based reference affinity work defined hierarchical reference affinity and tested two programs using x-means and k-means clustering [Zhong et al., 2003b].

Chilimbi et al. split Java classes into cold and hot portions according to their reference frequency [Chilimbi et al., 1999a]. Chilimbi later improved structure splitting using the frequency of data sub-streams called hot-streams [Chilimbi, 2001]. Hot-streams combines dynamic affinity with frequency but does not yet give whole-program reference affinity and requires a time-consuming profiling process. Rabbah and Palem gave another method for structure splitting. It finds opportunities for complete splitting by calculating the *neighbor affinity probability* without constructing an explicit affinity graph [Rabbah and Palem, 2003]. The probability shows the quality of a given layout but does not suggest the best reorganization.

Reference affinity may change during a dynamic execution. Researchers have examined various methods for dynamic data reorganization [Das et al., 1992; Ding and Kennedy, 1999a; Han and Tseng, 2000b; Mellor-Crummey et al., 2001; Mitchell et al., 1999; Strout et al., 2003]. Ding and Kennedy found that consecutive packing (first-touch data ordering) best exploits reference affinity for programs with good temporal locality [Ding and Kennedy, 1999a], an observation later confirmed by Mellor-Crummey et al. [Mellor-Crummey et al., 2001] and Strout et al [Strout et al., 2003]. Ding and Kennedy considered the time distance of data reuses and used the information in group packing. They also gave a unified model in which consecutive packing and group packing became special cases. In principle, the model of reference affinity can be used at run time to analyze sub-parts of an execution. However, it must be very efficient to be cost effective. In Chapter 4, the distance-based analysis is applied to phase-based data reorganization.

Locality between multiple arrays can be improved by array padding [Bailey, 1992; Rivera and Tseng, 1998], which changes the space between arrays or columns of arrays to reduce cache conflicts. In comparison, data regrouping is preferable because it works for all sizes of arrays on all configurations of cache, but padding is still needed if not all arrays can be grouped together.

The distance-based affinity model is the first trace-based model of hierarchical data locality providing strict properties. The distance-based affinity analysis finds array and structure field organization (among an exponential number of choices) that is consistently better than the layout given by the programmer, compiler, or statistical clustering. The interprocedure lightweight analysis presents the first practical interprocedure affinity analysis.

3.7 Future Directions

The locality prediction presented in last chapter and affinity models in this chapter all targeted traditional sequential programs. A possible extension is to make those techniques applicable to a wider range of applications, such as parallel programs and object-oriented programs running on virtual machines with garbage collectors, and a larger class of architectures, including Symmetric Multiprocessors (SMP) and Chip Multiprocessors (CMP).

A difficulty of parallel programs is to obtain accurate data reference traces. An instrumented parallel program usually won't have the same schedule as the original one, which may deviate the observed access sequence from the original. Among the few efforts to measure locality of parallel programs, people either used simulators [Faroughi, 2005; Forney et al., 2001] or instrumentors with the perturbation caused by the instrumented code ignored [Kim et al., 1999]. It is yet unclear how the perturbation and the simulator affect the accuracy of locality measurement. It remains an open question to obtain accurate data reference traces of a parallel program running on a real machine.

CMP is becoming the trend of the future computers, which raises the urgency of a better understanding and prediction of the shared-cache performance. Chandra et al. proposed a statistical scheme to predict the inter-thread cache contention on a CMP based on the stack distance profile of each thread's exclusive execution on the chip [Chandra et al., 2005]. Although they showed less than 4% average prediction error, their model cannot predict the behavior on any input other than the training one.

The garbage collector in Java-like programs provides a handy way to manipulate memory objects during run-time. It eases the locality optimization, but on the other hand, those programs tend to have a large number of objects and branches. The garbage collection itself introduces new memory problems. It remains an open question to effectively characterize and optimize locality of those programs online.

3.8 Summary

Affinity analysis is an effective tool for data layout transformations. This chapter describes a hierarchical affinity model, and two analysis techniques through profiling and an intereprocedural analysis respectively. The lightweight technique has been tested in a production compiler and has demonstrated significant performance improvement through array regrouping. The result suggests that array regrouping is an excellent candidate for inclusion in future optimizing compilers.

The locality and affinity models discussed in the prior chapters treat a program's execution as a whole. However, a program usually consists of more than one phases. A compiler, for example, includes parsing, loop optimization, register coloring and other steps. Different phases likely have different behavior. It would be desirable to detect those phases, predict their behavior, and dynamically adapt the program or running environment accordingly, which is the next topic of this thesis.

4 Locality Phase Analysis through Wavelet Transform

As computer memory hierarchy becomes adaptive, its performance increasingly depends on forecasting the dynamic program locality. This paper presents a method that predicts the locality phases of a program by a combination of locality profiling and run-time prediction. By profiling a training input, it identifies locality phases by sifting through all accesses to all data elements using variable-distance sampling, wavelet filtering, and optimal phase partitioning. It then constructs a phase hierarchy through grammar compression. Finally, it inserts phase markers into the program using binary rewriting. When the instrumented program runs, it uses the first few executions of a phase to predict all its later executions.

Compared with existing methods based on program code and execution intervals, locality phase prediction is unique because it uses locality profiles, and it marks phase boundaries in program code. The second half of the paper presents a comprehensive evaluation. It measures the accuracy and the coverage of the new technique and compares it with best known run-time methods. It measures its benefit in adaptive cache resizing and memory remapping. Finally, it compares the automatic analysis with manual phase marking. The results show that locality phase prediction is well suited for identifying large, recurring phases in complex programs.

4.1 Introduction

Memory adaptation is increasingly important as the memory hierarchy becomes deeper and more adaptive, and programs exhibit dynamic locality. To adapt, a program may reorganize its data layout multiple times during an execution. Several studies have examined dynamic data reorganization at the program level [Ding and Kennedy, 1999a; Han and Tseng, 2000a; Mellor-Crummey et al., 2001; Pingali et al., 2003; Strout et al., 2003] and at the hardware level [Luk and Mowry, 1999; Zhang et al., 2001]. They showed impressive improvements in cache locality and prefetching efficiency. Unfortunately, these techniques are not yet widely used partly because they need manual analysis to find program phases that benefit from memory adaptation. In this chapter, we show that this problem can be addressed by locality-based phase prediction.

Following early studies in virtual memory management by Batson and Madison [Batson and Madison, 1976] and by Denning [Denning, 1980], we define a locality phase as a period of a program execution that has stable or slow changing data locality inside the phase but disruptive transition periods between phases. [1] For optimization purpose, we are interested in phases that are repeatedly executed with similar locality. While data locality is not easy to define, we use a precise measure in this paper. For an execution of a phase, we measure the locality by its miss rate across all cache sizes and its number of dynamic instructions. At run time, phase prediction means knowing a phase and its locality whenever the execution enters the phase. Accurate prediction is necessary to enable large-scale memory changes while avoiding any adverse effects.

Many programs have recurring locality phases. For example, a simulation program may test the aging of an airplane model. The computation sweeps through the mesh structure of the airplane repeatedly in many time steps. The cache behavior of each

---

[1] Note that different authors define "phase" in different ways. In this thesis, We use it to refer to a span of program execution whose behavior, while potentially very nonuniform, is *predictable* in some important respect, typically because it resembles the behavior of some other execution span. Some authors, particularly those interested in fine-grain architectural adaptation, define a phase to be an interval whose behavior is *uniform* in some important respect (e.g., instruction mix or cache miss rate).

time step should be similar because the majority of the data access is the same despite local variations in control flow. Given a different input, for example another airplane model or some subparts, the locality of the new simulation may change radically but it will be consistent within the same execution. Similar phase behavior are common in structural, mechanical, molecular, and other scientific and commercial simulations. These programs have great demand for computing resources. Because of their dynamic but stable phases, they are good candidates for adaptation, if we can predict locality phases.

We describe a new prediction method that operates in three steps. The first analyzes the data locality in profiling runs. By examining the distance of data reuses in varying lengths, the analysis can “zoom in" and “zoom out" over long execution traces and detects locality phases using *variable-distance sampling*, *wavelet filtering*, and *optimal phase partitioning*. The second step then analyzes the instruction trace and identifies the phase boundaries in the code. The third step uses grammar compression to identify phase hierarchies and then inserts program markers through binary rewriting. During execution, the program uses the first few instances of a phase to predict all its later executions. The new analysis considers both program code and data access. It inserts static markers into the program binary without accessing the source code.

Phase prediction has become a focus of much recent research. Most techniques can be divided into two categories. The first is interval based. It divides a program execution into fixed-length intervals and predicts the behavior of future intervals from past observations. Interval-based prediction can be implemented entirely and efficiently at run time [Balasubramonian et al., 2000b, 2003; Dhodapkar and Smith, 2002, 2003; Duesterwald et al., 2003; Sherwood et al., 2003; Nagpurkar et al., 2006]. It handles arbitrarily complex programs and detects dynamically changing patterns. However, run-time systems cannot afford detailed data analysis much beyond counting the cache misses. In addition, it is unclear how to pick the interval length for different programs and for different inputs of the same program. The second category is code based. It marks a subset of loops and functions as phases and estimates their behavior through profiling [Hsu and Kremer, 2003; Huang et al., 2003; Magklis et al., 2003; Lau et al., 2006]. Pro-active rather than reactive, it uses phase markers to control the hardware and reduce the need for run-time monitoring. However, the program structure may not reveal its locality pattern. A phase may have many procedures and loops. The same procedure or loop may belong to different locality phases when accessing different data at different invocations. For example, a simulation step in a program may span thousands of lines of code with intertwined function calls and indirect data access.

In comparison, the new technique combines locality analysis and phase marking. The former avoids the use of fixed-size windows in analysis or prediction. The latter enables pro-active phase adaptation. In addition, the phase marking considers all instructions in the program binary in case the loop and procedure structures are obfuscated by an optimizing compiler.

In evaluation, we show that the new analysis finds recurring phases of widely varying sizes and nearly identical locality. The phase length changes in tune with program inputs and ranges from two hundred thousand to three billion instructions—this *length* is predicted with 99.5% accuracy. We compare it with other phase prediction methods, and we show its use in adaptive cache resizing and phase-based memory remapping.

Locality phase prediction is not effective on all programs. Some programs may not have predictable phases. Some phases may not be predictable from its data locality. We limit our analysis to programs that have large predictable phases, which nevertheless include important classes of dynamic programs. The next chapter describes another technique to tackle other input-sensitive programs as a compiler, transcoding utilities and a database.

4.2 Hierarchical Phase Analysis

This section first motivates the use of locality analysis and then describes the steps of locality-based phase prediction.

4.2.1 Locality Analysis Using Reuse Distance

In 1970, Mattson et al. defined the *LRU-stack distance* as the number of distinct data elements accessed between two consecutive references to the same element [Mattson et al., 1970]. They summarized the locality of an execution by the distance histogram, which determines the miss rate of fully-associative LRU cache of all sizes. Building on decades of development by others, Ding and Zhong reduced the analysis cost to near linear time. They found that reuse-distance histograms change in predictable patterns in large programs [Ding and Zhong, 2003]. In this work we go one step further to see whether predictable patterns exist for subparts of a program. For brevity, as mentioned in previous chapters, we call the LRU stack distance between two accesses of the same data the *reuse distance* of the latter access (to the previous access).

Reuse distance reveals patterns in program locality. We use the example of *Tomcatv*, a vectorized mesh generation program from SPEC95 known for its highly memory-sensitive performance. Figure 4.1 shows the reuse distance trace. Each data access is a point in the graph—the x-axis gives the logical time (i.e. the number of data accesses), and the y-axis gives the reuse distance [2]. The points are so numerous that they emerge as solid blocks and lines.

The reuse distance of data access changes continuously throughout the trace. We define a phase change as an abrupt change in data reuse pattern. In this example, the abrupt changes divide the trace into clearly separated phases. The same phases repeat in a fixed sequence. Reading the code documentation, we see indeed that the program

---

[2] To reduce the size of the graph, we show the reuse distance trace after variable-distance sampling described in Section 4.2.2.1.

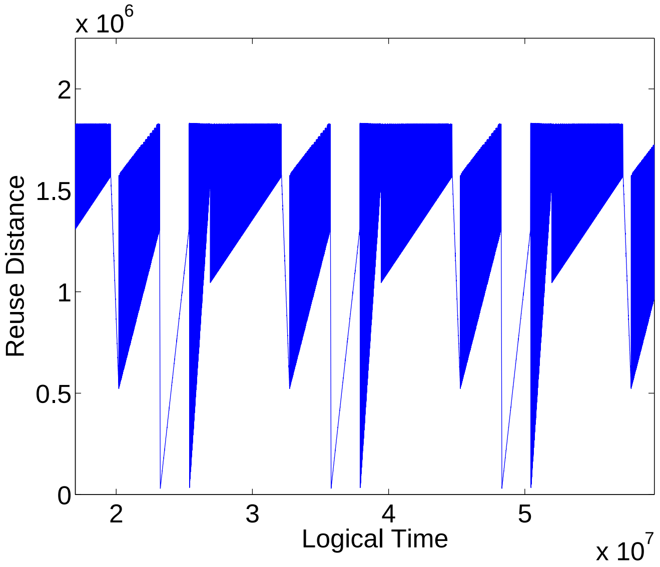


Figure 4.1: The reuse-distance trace of Tomcatv

has a sequence of time steps, each has five sub-steps—preparation of data, residual values, solving two tridiagonal systems, and adding corrections. What is remarkable is that we could see the same pattern from the reuse distance trace without looking at the program.

The example confirms four commonly held assumptions about program locality. First, the data locality may change constantly in an execution; however, major shifts in program locality are marked by radical rather than gradual changes. Second, locality phases have different lengths. The size of one phase has little relation with the size of others. Third, the size changes greatly with program inputs. For example, the phases of *Tomcatv* contain a few hundred million memory accesses in a training run but over twenty-five billion memory accesses in a test run. Finally, a phase often recurs with similar locality. *A phase is a unit of repeating behavior rather than a unit of uniform behavior.* To exploit these properties, locality phase prediction uses reuse distance to track fine-grain changes and find precise phase boundaries. It uses small training runs to predict larger executions.

Reuse distance measures locality better than pure program or hardware measures. Compiler analysis cannot fully analyze locality in programs that have dynamic data structures and indirect data access. The common hardware measure, the miss rate, is defined over a window. Even regular programs may have irregular cache miss rate distributions when we cut them into windows, as shown later in Figure 4.4. It is difficult to find a fixed window size that matches the phases of unequal lengths. We may use the miss trace, but a cache miss is a binary event—hit or miss for a given cache configuration. In comparison, reuse distance is a precise scale. It is purely a program property, independent of hardware configurations.

Reuse distance shows an interesting picture of program locality. Next we present a system that automatically uncovers the hierarchy of locality phases from this picture.

4.2.2 Off-line Phase Detection

Given the execution trace of training runs, phase detection operates in three steps: variable-distance sampling collects the reuse distance trace, wavelet filtering finds abrupt changes, and finally, optimal phase partitioning locates the phase boundary.

4.2.2.1 Variable-Distance Sampling

Instead of analyzing all accesses to all data, we sample a small number of representative data. In addition, for each data, we record only long-distance reuses because they reveal global patterns. Variable-distance sampling is based on the distance-based sampling described by Ding and Zhong [Ding and Zhong, 2003]. Their sampler uses ATOM to generate the data access trace and monitors the reuse distance of every access. When the reuse distance is above a threshold (the *qualification threshold*), the accessed memory location is taken as a data sample. A later access to a data sample is recorded as an access sample if the reuse distance is over a second threshold (the *temporal threshold*). To avoid picking too many data samples, it requires that a new data sample to be at least a certain space distance away (the *spatial threshold*) in memory from existing data samples.

The three thresholds in Ding and Zhong's method are difficult to control. Variable-distance sampling solves this problem by using dynamic feedback to find suitable thresholds. Given an arbitrary execution trace, its length, and the target number of samples, it starts with an initial set of thresholds. It periodically checks whether the rate of sample collection is too high or too low considering the target sample size. It changes the thresholds accordingly to ensure that the actual sample size is not far greater than the target. Since sampling happens off-line, it can use more time to find appropriate thresholds. In practice, variable-distance sampling finds 15 thousand to 30 thousand samples in less than 20 adjustments of thresholds. It takes several hours for the later steps of wavelet filtering and optimal phase partitioning to analyze these samples, although the long time is acceptable for our off-line analysis and can be improved by a more efficient implementation (currently using Matlab and Java).

The variable-distance sampling may collect samples at an uneven rate. Even at a steady rate, it may include partial results for executions that have uneven reuse density. However, the target sample size is large. The redundancy ensures that these samples together contain elements in all phase executions. If a data sample has too few access samples to be useful, the next analysis step will remove them as noise.

4.2.2.2 Wavelet Filtering

Viewing the sample trace as a signal, we use the *Discrete Wavelet Transform (DWT)* as a filter to expose abrupt changes in the reuse pattern. The DWT is a common technique in signal and image processing [Daubechies, 1992]. It shows the change of frequency over time. As a mutli-resolution analysis, the DWT applies two functions to data: the scale function and the wavelet function. The first smooths the signal by averaging its values in a window. The second calculates the magnitude of a range of frequencies in the window. The window then shifts through the whole signal. After finishing the calculations on the whole signal, it repeats the same process at the next level on the scaled results from the last level instead of on the original signal. This process may continue for many levels as a multi-resolution process. For each point on each level, a scaling and a wavelet coefficient are calculated using the variations of the following basic formulas:

$$c_j(k) = < f(x),\ 2^{-j}\phi(2^{-j}x - k) >$$
$$w_j(k) = < f(x),\ 2^{-j}\psi(2^{-j}x - k) >$$

where, $< a, b >$ is the scaler product of a and b, $f(x)$ is the input signal, j is the analysis level, ϕ and ψ are the scaling and wavelet function respectively. Many different wavelet families exist in the literature, such as *Haar, Daubechies,* and *Mexican-hat.* We use *Daubechies-6* in our experiments. Other families we have tested produce a similar result. On high-resolution levels, the points with high wavelet coefficient values signal abrupt changes; therefore they are likely phase changing points.

The wavelet filtering takes the reuse-distance trace of each data sample as a signal, then computes the level-1 coefficient for each access and removes from the trace the accesses with a low wavelet coefficient value. An access is kept only if its coefficient $\omega > m + 3\delta$, where m is the mean and δ is the standard deviation. The difference between this coefficient and others is statistically significant. We have experimented with coefficients of the next four levels and found the level-1 coefficient adequate.

Figure 4.2 shows the wavelet filtering for the access trace of a data sample in *MolDyn*, a molecular dynamics simulation program. The filtering removes accesses during the gradual changes because they have low coefficients. Note that it correctly removes accesses that correspond to local peaks. The remaining four accesses indicate global phase changes.

Sherwood et al. used the Fourier transform to find periodic patterns in execution trace [Sherwood et al., 2001]. The Fourier transform shows the frequencies appeared during the whole signal. In comparison, wavelets gives the *time-frequency* or the frequencies appeared over time. Joseph et al. used wavelets to analyze the change of processor voltage over time and to make on-line predictions using an efficient Haar-wavelet implementation [Joseph et al., 2004]. We use wavelets similar to their off-line analysis but at much finer granularity (because of the nature of our problem). Instead of filtering the access trace of all data, we analyze the sub-trace for each data element. This is critical because a gradual change in the subtrace may be seen as an abrupt change in the whole trace and cause false positives in the wavelet analysis. We will show an example later in Figure 4.4 (b), where most abrupt changes seen from the whole trace are not phase changes.

After it filters the sub-trace of each data sample, the filtering step recompiles the remaining accesses of all data samples in the order of logical time. The new trace is called a *filtered trace*. Since the remaining accesses of different data elements may signal the same phase boundary, we use optimal phase partitioning to further remove these redundant indicators.

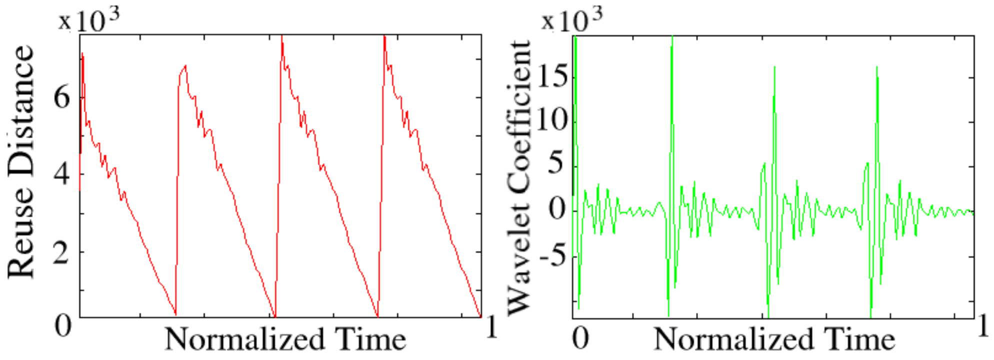


Figure 4.2: A wavelet transform example, where gradual changes are filtered out

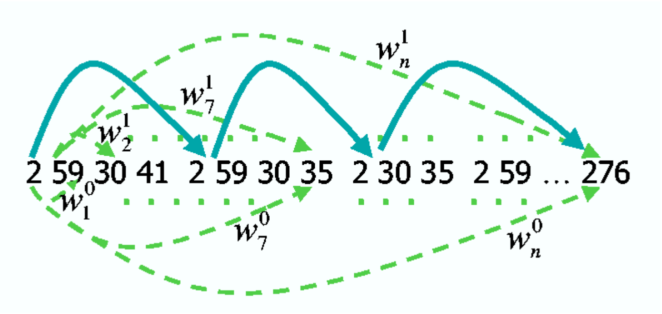


Figure 4.3: An example illustrating the optimal phase partitioning. Each number in the sequence represents the reference to a memory location. Notation $w_k{}^i$ represents the weight of the edge from the ith number to the kth. The solid lines show a path from the beginning to the end of the sequence.

4.2.2.3 Optimal Phase Partitioning

At a phase boundary, many data change their access patterns. Since the wavelet filtering removes reuses of the same data within a phase, the remaining is mainly accesses to different data samples clustered at phase boundaries. These two properties suggest two conditions for a good phase partition. First, a phase should include accesses to as many data samples as possible. This ensures that we do not artificially cut a phase into smaller pieces. Second, a phase should not include multiple accesses of the same data sample, since data reuses indicate phase changes in the filtered trace. The complication, however, comes from the imperfect filtering by the wavelet transform. Not all reuses represent a phase change.

We convert the filtered trace into a directed acyclic graph where each node is an access in the trace. Each node has a directed edge to all succeeding nodes as shown in Figure 4.3. Each edge (from access a to b) has a weight defined as $w_b{}^a = \alpha r + 1$, where $1 \geq \alpha \geq 0$, and r is the number of node recurrences between a and b. For example, the trace $aceefgefbd$ has two recurrences of e and one recurrence of f between c and b, so the edge weight between the two nodes is $3\alpha + 1$.

Intuitively, the weight measures how fit the segment from a to b is as a phase. The two factors in the weight penalize two tendencies. The first is the inclusion of reuses, and the second is the creation of new phases. For a sequence of memory references, the optimal case is a minimal number of phases with least reuses in each phase. Since the trace is not perfect, the weight and the factor α control the relative penalty for too large or too small phases. If α is 1, we prohibit any reuses in a phase. We may have as many phases as the length of the filtered trace. The result when $\alpha \geq 1$ is the same as $\alpha = 1$. If α is 0, we get one phase. In experiments, we found that the phase partitions were similar when α is between 0.2 and 0.8, suggesting that the noise in the filtered trace was acceptable. We used $\alpha = 0.5$ in the evaluation.

Once α is determined, shortest-path analysis on the directed graph finds a phase partition that minimizes the total penalty. It adds two nodes: a source node that has directed edges flowing to all original nodes, and a sink node that has directed edges coming from all original nodes. Any directed path from the source to the sink gives a phase partition. The sum of the weights of the edges on the path is called the *path weight*, showing the penalty of the phase partition. The best phase partition gives the least penalty, and it is given by the shortest path between the source and the sink.

Summary of off-line phase detection The program locality is a product of all accesses to all program data. The phase detection first picks enough samples in time and space to capture the high-level pattern. Then it uses wavelets to remove the temporal redundancy and phase partitioning to remove the spatial redundancy. The next challenge is marking the phases in program code. The wavelet filtering loses accurate time information because samples are considered a pair at a time (to measure the difference). In addition, the locality may change through a transition period instead of a transition point. Hence the exact time of a phase change is difficult to attain. We address this problem in the next step.

4.2.3 Phase Marker Selection

The instruction trace of an execution is recorded at the granularity of basic blocks. The result is a block trace, where each element is the label of a basic block. This step finds the basic blocks in the code that uniquely mark detected phases. Previous program analysis considered only a subset of code locations, for example function and loop boundaries [Hsu and Kremer, 2003; Huang et al., 2003; Magklis et al., 2003]. Our analysis examines all instruction blocks, which is equivalent to examining all program instructions. This is especially important at the binary level, where the high level program structure may be lost due to aggressive compiler transformations such as procedure in-lining, software pipelining, loop fusion, and code compression.

As explained earlier, phase detection finds the number of phases but cannot locate the precise time of phase transitions. The precision is in the order of hundreds of memory accesses while a typical basic block has fewer than ten memory references. Moreover, the transition may be gradual, and it is impossible to locate a single point. We solve this problem by using the frequency of the phases instead of the time of their transition.

We define the frequency of a phase by the number of its executions in the training run. Given the frequency found by the last step, we want to identify a basic block that is always executed at the beginning of a phase. We call it the *marker block* for this phase. If the frequency of a phase is f, the marker block should appear no more than f times in the block trace. The first step of the marker selection filters the block trace and keeps only blocks whose frequency is no more than f. If a loop is a phase, the filtering will remove the occurrences of the loop body block and keep only the header and the exit blocks. If a set of mutual recursive functions forms a phase, the filtering will remove the code of the functions and keep only the ones before and after the root invocation. After filtering, the remaining blocks are candidate markers.

After frequency-based filtering, the removed blocks leave large blank regions between the remaining blocks. If a blank region is larger than a threshold, it is considered as a phase execution. The threshold is determined by the length distribution of the blank regions, the frequency of phases, and the execution length. Since the training runs had at least 3.5 million memory accesses, we simply used 10 thousand instructions as the threshold. In other words, a phase execution must consume at least 0.3% of the total execution to be considered significant. We can use a smaller threshold to find sub-phases after we find large phases.

Once the phase executions are identified, the analysis considers the block that comes after a region as markers marking the boundary between the two phases. Two regions are executions of the same phase if they follow the same code block. The analysis picks markers that mark most if not all executions of the phases in the training run. We have considered several improvements that consider the length of the region, use multiple markers for the same phase, and correlate marker selection across multiple runs. However, this basic scheme suffices for programs we tested.

Requiring the marker frequency to be no more than the phase frequency is necessary but not sufficient for phase marking. A phase may be fragmented by infrequently executed code blocks. However, a false marker cannot divide a phase more than f times. In addition, the partial phases will be regrouped in the next step, phase-hierarchy construction.

4.2.4 Marking Phase Hierarchy

Hierarchical construction Given the detected phases, we construct a phase hierarchy using grammar compression. The purpose is to identify composite phases and increase the granularity of phase prediction. For example, for the *Tomcatv* program showed in Figure 4.1, every five phase executions form a time step that repeats as a composite phase. By constructing the phase hierarchy, we find phases of the largest granularity.

We use SEQUITUR, a linear-time and linear-space compression method developed by Nevill-Manning and Witten [Nevill-Manning and Witten, 1997]. It compresses a string of symbols into a Context Free Grammar. To build the phase hierarchy, we have developed a novel algorithm that extracts phase repetitions from a compressed grammar and represents them explicitly as a regular expression. The algorithm recursively converts non-terminal symbols into regular expressions. It remembers previous results so that it converts the same non-terminal symbol only once. A merge step occurs for a non-terminal once its right-hand side is fully converted. Two adjacent regular expressions are merged if they are equivalent (using for example the equivalent test described by Hopcroft and Ullman [Hopcroft and Ullman, 1979]).

SEQUITUR was used by Larus to find frequent code paths [Larus, 1999] and by Chilimbi to find frequent data-access sequences [Chilimbi, 2001]. Their methods model the grammar as a DAG and finds frequent sub-sequences of a given length. Our method traverses the non-terminal symbols in the same order, but instead of finding sub-sequences, it produces a regular expression.

Phase marker insertion The last step uses binary rewriting to insert markers into a program. The basic phases (the leaves of the phase hierarchy) have unique markers in the program, so their prediction is trivial. To predict the composite phases, we insert a predictor into the program. Based on the phase hierarchy, the predictor monitors the program execution and makes predictions based on the on-line phase history. Since the hierarchy is a regular expression, the predictor uses a finite automaton to recognize the current phase in the phase hierarchy. In the programs we tested so far, this simple method suffices. The cost of the markers and the predictor is negligible because they are invoked once per phase execution, which consists of on average millions of instructions as shown in the evaluation.

4.3 Evaluation

We conduct four experiments. We first measure the granularity and accuracy of phase prediction. We then use it in cache resizing and memory remapping. Finally, we test it against manual phase marking. We compare with other prediction techniques in the first two experiments.

Our test suite is given in Table 4.1. We pick programs from different sets of commonly used benchmarks to get an interesting mix. They represent common computation tasks in signal processing, combinatorial optimization, structured and unstructured mesh and N-body simulations, a compiler, and a database. *FFT* is a basic implementation from a textbook. The next six programs are from SPEC: three are floating-point and three are integer programs. Three are from SPEC95 suite, one from SPEC2K, and two (with small variation) are from both. Originally from the CHAOS group at University of Maryland, *MolDyn* and *Mesh* are two dynamic programs whose data access pattern depends on program inputs and changes during execution [Das et al., 1994]. They are commonly studied in dynamic program optimization [Ding and Kennedy, 1999a; Han and Tseng, 2000a; Mellor-Crummey et al., 2001; Strout et al., 2003]. The floating-point programs from SPEC are written in Fortran, and the integer programs are in C. Of the two dynamic programs, *MolDyn* is in Fortran, and *Mesh* is in C. We note that the choice of source-level languages does not matter because we analyze and transform programs at the binary level.

For programs from SPEC, we use the *test* or the *train* input for phase detection and the *ref* input for phase prediction. For the prediction of *Mesh*, we used the same mesh as that in the training run but with sorted edges. For all other programs, the prediction is tested on executions hundreds times longer than those used in phase detection.

We use ATOM to instrument programs to collect the data and instruction trace on a Digital Alpha machine [Srivastava and Eustace, 1994]. All programs are compiled

Table 4.1: Benchmarks for locality phase analysis

Benchmark	Description	Source
FFT	fast Fourier transformation	textbook
Applu	solving five coupled nonlinear PDE's	Spec2KFp
Compress	common UNIX compression utility	Spec95Int
Gcc	GNU C compiler 2.5.3	Spec95Int
Tomcatv	vectorized mesh generation	Spec95Fp
Swim	finite difference approximations for shallow water equation	Spec95Fp
Vortex	an object-oriented database	Spec95Int
Mesh	dynamic mesh structure simulation	CHAOS
MolDyn	molecular dynamics simulation	CHAOS

by the Alpha compiler using "-O5" flag. After phase analysis, we again use ATOM to insert markers into programs.

4.3.1 Phase Prediction

We present results for all programs except for *Gcc* and *Vortex*, which we discuss at the end of this section. We first measure the phase length and then look at the phase locality in detail.

Table 4.2 shows two sets of results. The upper half shows the accuracy and coverage of strict phase prediction, where we require that phase behavior repeats exactly including its length. Except for *MolDyn*, the accuracy is perfect in all programs, that is, *the number of the executed instructions is predicted exactly at the beginning of a phase execution*. We measure the coverage by the fraction of the execution time spent in the predicted phases. The high accuracy requirement hurts coverage, which is over 90% for four programs but only 46% for *Tomcatv* and 13% for *MolDyn*. If we relax the accuracy requirement, then the coverage increases to 99% for five programs and 98% and 93% for the other two, as shown in the lower half of the table. The accuracy drops to 90% in *Swim* and 13% in *MolDyn*. *MolDyn* has a large number of uneven phases when it

Table 4.2: Accuracy and coverage of phase prediction

Benchmarks	Strict accuracy		Relaxed accuracy	
	Accuracy (%)	Coverage (%)	Accuracy (%)	Coverage (%)
FFT	100	96.41	99.72	97.76
Applu	100	98.89	99.96	99.70
Compress	100	92.39	100	93.28
Tomcatv	100	45.63	99.9	99.76
Swim	100	72.75	90.16	99.78
Mesh	100	93.68	100	99.58
MolDyn	96.47	13.49	13.27	99.49
Avgerage	99.50	73.32	86.14	98.48

finds neighbors for each particle. In all programs, the phase prediction can attain either perfect accuracy, full coverage, or both.

The granularity of the phase hierarchy is shown in Table 4.3 and Table 4.4 by the average size of the smallest (leaf) phases and the largest composite phases. The left half shows the result of the detection run, and the right half shows the prediction run. The last row shows the average across all programs. With the exception of *Mesh*, which has two same-length inputs, the prediction run is larger than the detection run by, on average, 100 times in execution length and 400 times in the phase frequency. The average size of the leaf phase ranges from two hundred thousand to five million instructions in the detection run and from one million to eight hundred million in the prediction run. The largest phase is, on average, 13 times the size of the leaf phase in the detection run and 50 times in the prediction run.

The results show that the phase length is anything but uniform. The prediction run is over 1000 times longer than the detection run for *Applu* and *Compress* and nearly 5000 times longer for *MolDyn*. The longer executions may have about 100 times more phase executions (*Tomcatv*, *Swim*, and *Applu*) and over 1000 times larger phase size (in *Compress*). The phase size differs from phase to phase, program to program, and input

Table 4.3: Number and the size of phases in detection runs

Tests	leaf phases	exe. len. (M inst.)	avg. leaf size (M inst.)	avg. largest phase size (M inst.)
FFT	14	23.8	2.5	11.6
Applu	645	254.3	0.394	3.29
Compress	52	52.0	0.667	2.2
Tomcatv	35	175.0	4.9	34.9
Swim	91	376.7	4.1	37.6
Mesh	4691	5151.9	1.1	98.2
MolDyn	59	11.9	0.202	3.97
Average	798	863.66	1.98	27.39

Table 4.4: Number and the size of phases in prediction runs

Tests	leaf phases	exe. len. (M inst.)	avg. leaf size (M inst.)	avg. largest phase size (M inst.)
FFT	122	5730.4	50.0	232.2
Applu	4437	335019.8	75.5	644.8
Compress	52	62418.4	800.2	2712.0
Tomcatv	5250	24923.2	4.7	33.23
Swim	8101	33334.9	4.1	37.03
Mesh	4691	5151.9	1.1	98.2
MolDyn	569	50988.1	89.6	1699.6
Average	3317	73938.1	146.5	779.58

to input, suggesting that a single interval or threshold would not work well for this set of programs.

4.3.1.1 Comparison of Prediction Accuracy

Figure 4.4 shows the locality of two representative programs—*Tomcatv* and *Compress*—in two columns of three graphs each. The upper graphs show the phase detection in training runs. The other graphs show phase prediction in reference runs. The upper graphs show a fraction of the sampled trace with vertical lines marking the phase boundaries found by variable-distance sampling, wavelet filtering, and optimal phase partitioning. The lines fall exactly at the points where abrupt changes of reuse behavior happen, showing the effect of these techniques. The phases have different lengths. Some are too short in relative length and the two boundaries become a single line in the graph. The numbers next to the lines are the basic block IDs where markers are inserted. The same code block precedes and only precedes the same locality phase, showing the effect of marker selection.

The middle two graphs show the locality of predicted phases. To visualize the locality, we arbitrarily pick two different cache sizes—32KB and 256KB cache—and use the two miss rates as coordinates. Each execution of a phase is a cross (X) on the graph. *Tomcatv* has 5251 executions of 7 locality phases: all five thousand crosses are mapped to seven in the graph. Most crosses overlap perfectly. The phase prediction is correct in all cases because the executions of the same phase maps to a single cross except for a small difference in the second and third phase, where the first couple of executions have slightly different locality. We label each phase by the phase ID, the relative frequency, and the range of phase length. The relative frequency is the number of the executions of a phase divided by the total number of phase executions (5251 for *Tomcatv*). The last two numbers give the number of instructions in the shortest and the longest execution of the phase, in the unit of millions of instructions. *Compress* is shown by the same format. It has 52 executions of 4 locality phases: all 52 crosses map to four, showing perfect prediction accuracy. The phase length ranges from 2.9 thousand to 1.9 million instructions in two programs. For each phase, the length prediction is accurate to at least three significant digits.

The power of phase prediction is remarkable. For example, in *Compress*, when the first marker is executed for the second time, the program knows that it will execute 1.410 million instructions before reaching the next marker, and that the locality is the same for every execution. This accuracy confirms our assumption that locality phases are marked by abrupt changes in data reuse.

Phase vs. interval An interval method divides the execution into fixed-size intervals. The dots in the bottom graphs of Figure 4.4 show the locality of ten million

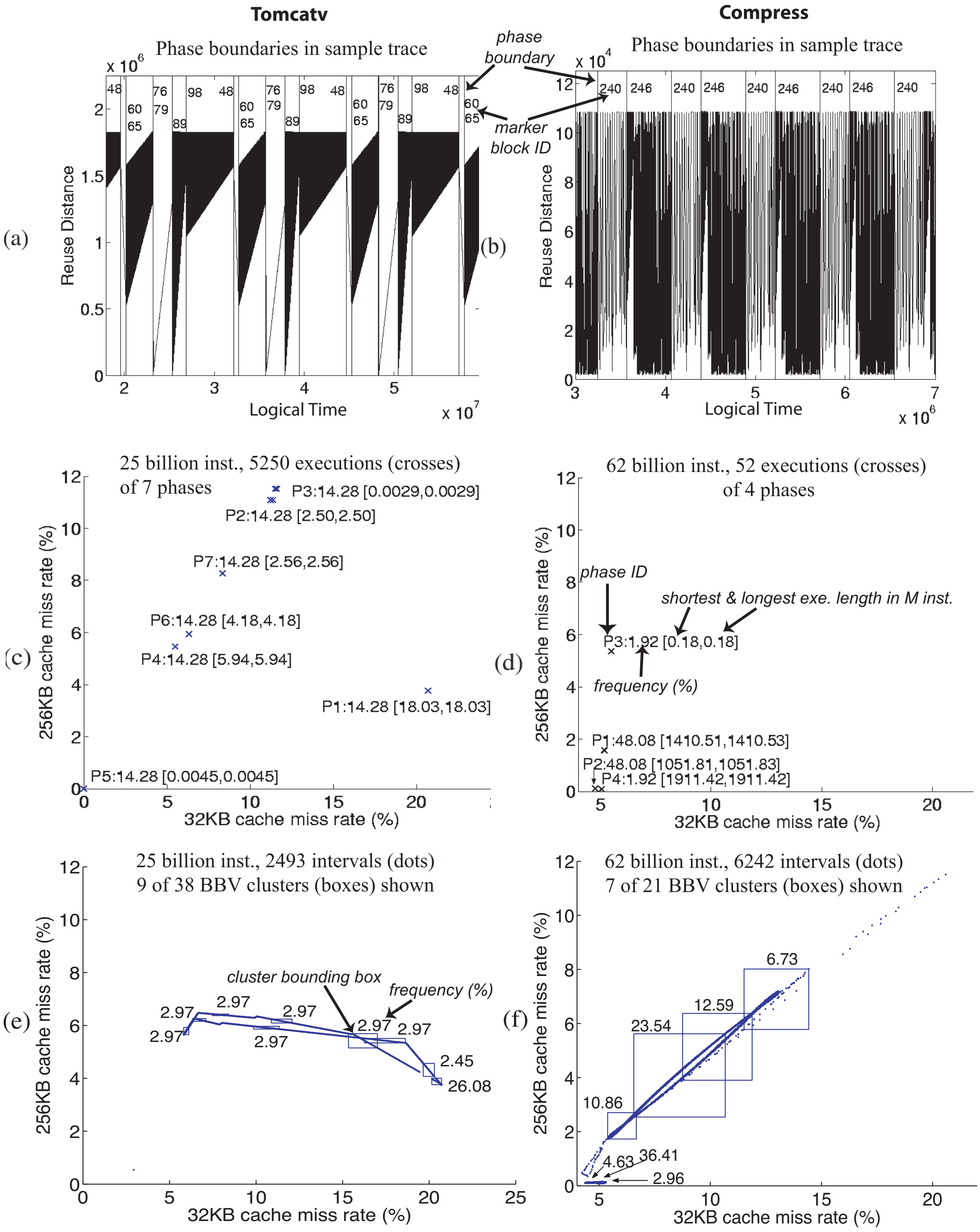


Figure 4.4: Prediction Accuracy for *Tomcatv* and *Compress*. Part (a) and (b) show the phase boundaries found by off-line phase detection. Part (c) and (d) show the locality of the phases found by run-time prediction. As a comparison, Part (e) and (f) show the locality of ten million-instruction intervals and BBV (basic-block vector) clusters.

instruction intervals. The 2493 dots in *Tomcatv* and 6242 dots in *Compress* do not suggest a regular pattern.

Both the phases and intervals are partitions of the same execution sequence—the 25 billion instructions in *Tomcatv* and 62 billion in *Compress*. Yet the graphs are a striking contrast between the sharp focus of phase crosses and the irregular spread of interval dots—it indeed matters where and how to partition an execution into phases. Locality phases are selected at the right place with the right length, while intervals are a uniform cut. Compared to the phases, the intervals are too large to capture the two to four million-instruction phases in *Tomcatv* and too small to find the over one billion-instruction phases in *Compress*. While the program behavior is highly regular and fully predictable for phases, it becomes mysteriously irregular once the execution is cut into intervals.

Phase vs. BBV Three major phase analysis techniques have been examined [Dhodapkar and Smith, 2003]—procedure-based [Huang et al., 2003; Magklis et al., 2003], code working set [Dhodapkar and Smith, 2002], and basic-block vector (BBV) [Sherwood et al., 2003]. By testing the variation in IPC (instruction per cycle), it concluded that BBV is the most accurate. We implemented BBV prediction according to the algorithm of Sherwood et al [Sherwood et al., 2003]. Our implementation uses the same ten million-instruction windows and the same threshold for clustering. We implemented their Markov predictor but in this section we use only the clustering (perfect prediction). It randomly projected the frequency of all code blocks into a 32-element vector before clustering. Instead of using IPC, we use locality as the metric for evaluation.

BBV clusters the intervals based on their code signature and execution frequency. We show each BBV cluster by a bounding box labeled with the relative frequency. BBV analysis produces more clusters than those shown. We do not show boxes for clusters whose frequency is less than 2.1%, partly to make the graph readable. We note that the aggregated size of the small clusters is quite large (51%) for *Tomcatv*. In addition, we exclude the outliers, which are points that are farthest from the cluster center (3δ, statistically speaking); otherwise the bounding boxes are larger.

As shown by previous studies [Dhodapkar and Smith, 2003; Sherwood et al., 2003], BBV groups intervals that have similar behavior. In *Tomcatv*, the largest cluster accounts for 26% of the execution. The miss rate varies by less than 0.3% for the 256KB cache and 0.5% for the 32KB cache. However, the similarity is not guaranteed. In the worst case in *Compress*, a cluster of over 23% execution has a miss rate ranging from 2.5% to 5.5% for the 256KB cache and from 7% to 11% for the 32KB cache. In addition, different BBV clusters may partially intersect. Note that with fine-tuned parameters we will see smaller clusters with lower variation. In fact, in the majority of cases in these programs, BBV produces tight clusters. However, even in best cases, BBV clusters do no have perfectly stacked points as locality phases do.

Table 4.5 shows the initial and normalized standard deviation. The locality is an 8-element vector that contains the miss rate for cache sizes from 32KB to 256KB in 32KB increments. The standard deviation is calculated for all executions of the same phase and the intervals of each BBV cluster. Then the standard deviation of all phases or clusters are averaged (weighted by the phase or cluster size) to produce the number for the program. The numbers of BBV clustering and prediction, shown by the last two columns, are similarly small as reported by Sherwood et al. for IPC [Sherwood et al., 2003]. Still, the numbers for locality phases are much smaller—one to five orders of magnitude smaller than that of BBV-based prediction.

So far we measure the cache miss rate through simulation, which does not include all factors on real machines such as that of the operating system. We now examine the L1 miss rate on an IBM Power 4 processor for the first two phases of *Compress* (the other two phases are too infrequent to be interesting). Figure 4.5 shows the measured miss rate for each execution of the two phases. All but the first execution of Phase 1 have nearly identical miss rates on the 32KB 2-way data cache. The executions of Phase 2 show more variation. The effect from the environment is more visible in Phase

Table 4.5: Standard deviation of locality phases and BBV phases

	standard deviations		
	locality phase prediction	BBV clustering	BBV RLE Markov prediction
FFT	6.87E-8	0.00040	0.0061
Applu	5.06E-7	2.30E-5	0.00013
Compress	3.14E-6	0.00021	0.00061
Tomcatv	4.53E-7	0.00028	0.0016
Swim	2.66E-8	5.59E-5	0.00018
Mesh	6.00E-6	0.00012	0.00063
MolDyn	7.60E-5	0.00040	0.00067

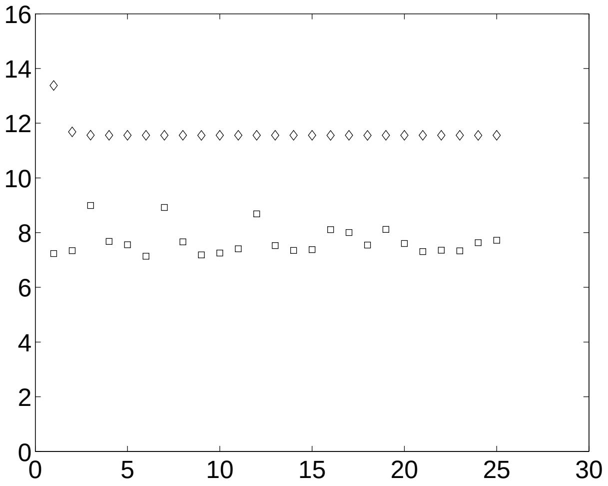


Figure 4.5: The miss rates of *Compress* phases on IBM Power 4

2 likely because its executions are shorter and the miss rate lower than those of the first phase.

The comparison with interval-based methods is partial because we use only programs that are amenable to locality-phase prediction. Many dynamic programs do not have consistent locality. For them interval-based methods can still exploit run-time patterns, while this scheme would not work because it assumes that each phase, once in execution, maintains identical locality. Next are two such examples.

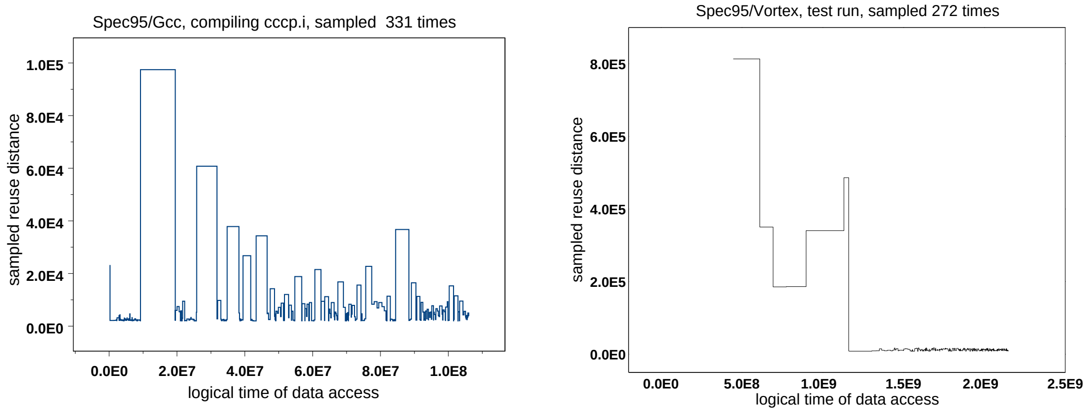


Figure 4.6: Sampled reuse distance trace of *Gcc* and *Vortex*. The exact phase length is unpredictable in general.

4.3.1.2 Gcc and Vortex

The programs *Gcc* and *Vortex* are different because their phase length is not consistent even in the same execution. In *Gcc*, the phase length is determined by the function being compiled. Figure 4.6 shows the distance-based sample trace. Unlike previous trace graphs, it uses horizontal steps to link sample points. The peaks in the upper graph roughly correspond to the 100 functions in the 6383-line input file. The size and location of the peaks are determined by the input and are not constant.

Vortex is an object-oriented database. The test run first constructs a database and then performs a set of queries. The lower figure of Figure 4.6 shows the sample trace. It shows the transition from data insertion to query processing. However, in other inputs, the construction and queries may come in any order. The exact behavior, like *Gcc*, is input dependent and not constant.

The next chapter discusses those programs in more details and proposes a technique based on active profiling to effectively detect phases in those programs.

4.3.2 Adaptive Cache Resizing

During an execution, cache resizing reduces the physical cache size without increasing the miss rate [Balasubramonian et al., 2000b; Huang et al., 2003]. Therefore, it can reduce the access time and energy consumption of the cache without losing performance. We use a simplified model where the cache consists of 64-byte blocks and 512 sets. It can change from direct mapped to 8-way set associative, so the cache size can change between 32KB and 256KB in 32KB units. In the adaptation, we need to predict the smallest cache size that yields the same miss rate as the 256KB cache.

As seen in the example of *Tomcatv*, program data behavior changes constantly. A locality phase is a unit of repeating behavior rather than a unit of uniform behavior. To capture the changing behavior inside a large phase, we divide it into 10K intervals (called phase intervals). The adaptation finds the best cache size for each interval during the first few executions and reuses them for later runs. The scheme needs hardware support but needs no more than that of interval-based cache resizing.

Interval-based cache resizing divides an execution into fixed-length windows. The basic scheme is to monitor some run-time behavior like branch miss prediction rate and instruction per cycle during the execution of an interval [Balasubramonian et al., 2000b]. If the difference between this and the last interval exceeds a threshold, the execution is considered as entering a brand new phase. The scheme doesn't categorize intervals into a number of phases, but only detects whether the current interval is similar enough to the last one. If not, it immediately starts a new exploration to find the best cache size for the new phase, even if many similar intervals have appeared before the last interval. Note that it's possible for a new phase to be too short for the exploration to finish. In that case, at the phase change, the incomplete exploration will be discarded and a new one will start. Once an exploration completes, the best cache configuration for the current phase is determined. The system then keeps using this configuration for the later intervals until it detects another phase change, which triggers a new exploration.

We test the basic interval-based scheme using five different interval lengths: 10K, 1M, 10M, 40M, and 100M memory accesses. In the experiment, we assume perfect phase detection: there is a phase change if the best cache size of the current interval differs from the last one. (We know the best cache size of each phase or interval by running it through *Cheetah*, a cache simulator that measures the miss rate of all eight cache sizes at the same time [Sugumar and Abraham, 1993].) The implementation is idealistic because in reality the interval-based scheme usually cannot detect all phase changes precisely, due to the imperfect correlation between the monitored run-time behavior and the optimal cache configurations. Moreover, the idealistic version doesn't have detection latency: the earliest time to detect a new phase in reality is after the termination of the first interval in that phase when the information of the hardware events of the interval becomes available. The upshot is that every interval uses its optimal cache configuration except the exploration periods in the idealistic scheme.

Some studies extend the basic interval-based scheme by considering code information such as code working set [Dhodapkar and Smith, 2002] and basic-block vector (BBV) [Sherwood et al., 2003]. We test a BBV predictor using 10M instruction windows, following the implementation of Sherwood et al [Sherwood et al., 2003] and uses a run-length encoding Markov predictor to predict the phase identity of the next interval (the best predictor reported in [Sherwood et al., 2003]). Unlike the basic interval-based scheme, the BBV method categorizes past intervals into a number of phases and assigns each phase an identity. At the beginning of an interval, the method predicts the interval's phase identity or assigns a new identity if this interval is very different from any previous ones. In our experiment, the implemented scheme is also idealistic: an exploration is triggered only when a new phase identity is assigned or the current interval has a different optimal cache configuration from the one obtained from the last exploration of its phase. In comparison, the basic interval-based scheme starts a new exploration at every point when the optimal configuration becomes different from the last interval.

For run-time exploration, we count the minimal cost—each exploration takes exactly two trial runs, one at the full cache size and one at the half cache size. Then we start using the best cache size determined by the oracle, the cache simulator *Cheetah*. The results for interval and BBV methods are idealistic. While the result of the locality-phase method is real; each phase uses the optimal cache configuration of its first instance for all its instances after the exploration period. Because it is able to precisely predict the exact behavior repetition, the locality-phase method can amortize the exploration cost over many executions. With the right hardware support, it can gauge the exact loss compared to the full size cache and guarantee a bound on the absolute performance loss.

Figure 4.7 shows the average cache size from locality-phase, interval, and BBV methods. The upper graph shows the results of adaptation with no miss-rate increase. The results are normalized to the phase method. The largest cache size, 256KB, is shown as the last bar in each group. Different intervals find different cache sizes, but all reductions are less than 10%. The average is 6%. BBV gives consistently good reduction with a single interval size. The improvement is at most 15% and on average 10%. In contrast, the phase adaptation reduces the cache size by 50% for most programs and over 35% on average.

The lower graph in Figure 4.7 shows the results of adaptation with a 5% bound on the miss-rate increase. The effect of interval methods varies greatly. The 10M interval was 20% better than the locality phase for *FFT* but a factor of three worse for *Tomcatv* and *Swim*. The 100M interval has the best average reduction of nearly 50%. BBV again shows consistently good reduction with a single interval size. On average it is slightly better than the best interval method. The phase method reduces the cache size more than other methods do for all programs except for *FFT*. *FFT* has varied behavior, which causes the low coverage and consequently not as large cache-size reduction by locality phase prediction. *MolDyn* does not have identical locality, so phase-based resizing causes a 0.6% increase in the number of cache misses. Across all programs, the average reduction using locality phases is over 60%.

Figure 4.8 shows cache miss rate increases due to the cache resizing. The upper graph shows 0 to 1.2% increase with at most 0.2% on average. The increase mainly comes from the indispensable explorations. The lower graph demonstrates less than 4.5% increase, satisfying the required 5% upper bound.

The effectiveness of locality phases is because of their accurate phase boundaries and the high consistency of phase behavior. Interval-based schemes including BBV method cannot work as well because the right phase boundaries may not match interval boundaries. The large variance of BBV phase instances, illustrated by Figure 4.4, incurs much more explorations per phase than those per locality phase.

Earlier studies used more accurate models of cache and measured the effect on time and energy through cycle-accurate simulation. Since simulating the full execution takes a long time, past studies either used a partial trace or reduced the program input size [Balasubramonian et al., 2000b; Huang et al., 2003]. We choose to measure the miss rate of full executions. While it does not give the time or energy, the miss rate is accurate and reproducible by others without significant efforts in calibration of simulation parameters.

4.3.3 Phase-Based Memory Remapping

We use locality phases in run-time memory remapping. To support dynamic data remapping at the phase boundary, we assume that the machine is equipped with the *Impulse* memory controller, developed by Carter and his colleagues at University of Utah [Zhang et al., 2001; Zhang, 2000]. *Impulse* reorganizes data without actually copying them to CPU or in memory. For example, it may create a column-major version

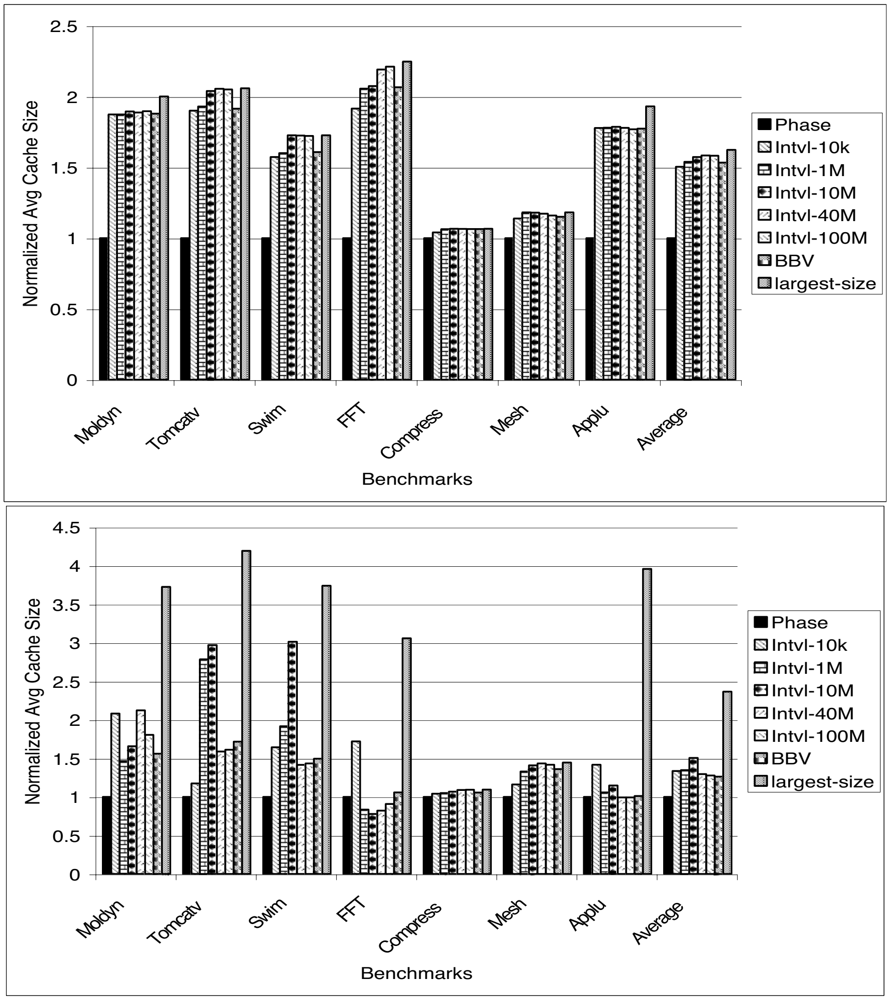


Figure 4.7: Average cache-size reduction by phase, interval, and BBV prediction methods, assuming perfect phase-change detection and minimal-exploration cost for interval and BBV methods. Upper graph: no increase in cache misses. Lower graph: at most 5% increase.

of a row-major array via remapping. A key requirement for exploiting *Impulse* is to identify the time when remapping is profitable.

We consider affinity-based array remapping, where arrays that tend to be accessed concurrently are interleaved by remapping [Zhong et al., 2004]. To demonstrate the value of locality phase prediction, we evaluate the performance benefits of redoing the remapping for each phase rather than once for the whole program during compilation.

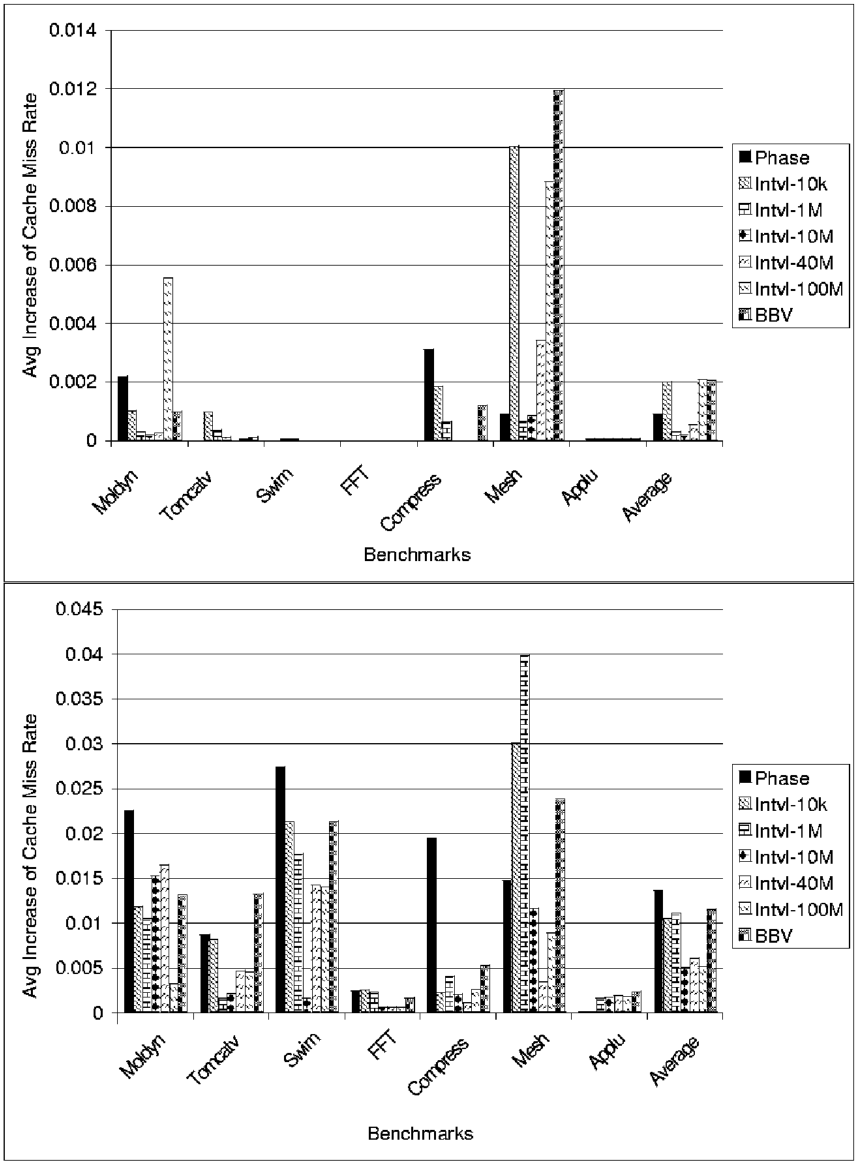


Figure 4.8: Average cache miss rate increase due to the cache resizing by phase, interval, and BBV prediction methods. Upper graph: the objective is no increase in cache misses. Lower graph: the objective is at most 5% increase.

We apply affinity analysis for each phase and insert remapping code at the location of the phase marker. The following table shows the execution time in seconds on 2GHz Intel Pentium IV machine with the *gcc* compiler using *-O3*.

Table 4.6: Performance improvement from phase-based array regrouping, excluding the cost of run-time data reorganization

Benchmark	Original	Phase (speedup)	Global (speedup)
Mesh	4.29	4.17 (2.8%)	4.27 (0.4%)
Swim	52.84	34.08 (35.5%)	38.52(27.1%)

For the two programs, we obtain speedups of 35.5% and 2.8% compared to the original program and 13% and 2.5% compared to the best static data layout [Zhong et al., 2004], as shown in Table 4.6. In the absence of an *Impulse* implementation, we program the remapping and calculate the running time excluding the remapping cost. Table 7.3 of Zhang's dissertation shows the overhead of setting up remappings for a wide range of programs. The overhead includes setting up shadow region, creating memory controller page table, data flushing, and possible data movement. The largest overhead shown is 1.6% of execution time for static index vector remapping [Zhang, 2000].

For example for the 14 major arrays in *Swim*, whole-program analysis shows close affinity between array *u* and *v*, *uold* and *pold*, and *unew* and *pnew*. Phase-based analysis shows affinity group {*u,v,p*} for the first phase, {*u,v,p,unew,vnew,pnew*} for the second phase, and three other groups, {*u,uold,unew*}, {*v,vold,vnew*}, and {*p,pold,pnew*}, for the third phase. Compared to whole-program reorganization, the phase-based optimization reduces cache misses by one third (due to array *p*) for the first phase, by two thirds for the second phase, and by half for the third phase.

Using the two example programs, we have shown that phase prediction finds opportunities of dynamic data remapping. The additional issues of affinity analysis and code transformation are discussed by Zhong et al [Zhong et al., 2004]. The exact interaction with *Impulse* like tools is a subject of future study.

4.3.4 Comparison with Manual Phase Marking

We hand-analyzed each program and inserted phase markers (*manual markers*) based on our reading of the code and its documentation as well as results from *gprof* (to find important functions). We compare manual marking with automatic marking as follows. As a program runs, all markers output the logical time (the number of memory accesses from the beginning). Given the set of logical times from manual markers and the set from auto-markers, we measure the overlap between the two sets. Two logical times are considered the same if they differ by no more than 400, which is 0.02% of the average phase length. We use the recall and precision to measure their closeness. They are defined by the formulas below. The recall shows the percentage of the manually marked times that are marked by auto-markers. The precision shows the percentage of the automatically marked times that are marked manually.

$$Recall = \frac{\mid M \cap A \mid}{\mid M \mid} \tag{4.1}$$

$$Precision = \frac{\mid M \cap A \mid}{\mid A \mid} \tag{4.2}$$

where M is the set of times from the manual markers, and A is the set of times from auto-markers.

Table 4.7 shows a comparison with manually inserted markers for detection and prediction runs. The columns for each run give the recall and the precision. The recall is over 95% in all cases except for *MolDyn* in the detection run. The average recall increases from 96% in the detection run to 99% in the prediction run because the phases with a better recall occur more often in longer runs. Hence, the auto-markers capture the programmer's understanding of the program because they catch nearly all manually marked phase changing points.

Table 4.7: Overlap with manual phase markers

Benchmark	Detection		Prediction	
	Recall	Prec.	Recall	Prec.
FFT	1	1	1	1
Applu	0.993	0.941	0.999	0.948
Compress	0.987	0.962	0.987	0.962
Tomcatv	0.952	0.556	1	0.571
Swim	1	0.341	1	0.333
Mesh	1	0.834	1	0.834
MolDyn	0.889	0.271	0.987	0.267
Average	0.964	0.690	0.986	0.692

The precision is over 95% for *Applu* and *Compress*, showing that automatic markers are effectively the same as the manual markers. *MolDyn* has the lowest recall of 27%. We checked the code and found the difference. When the program is constructing the neighbor list, the analysis marks the neighbor search for each particle as a phase while the programmer marks the searches for all particles as a phase. In this case, the analysis is correct. The neighbor search repeats for each particle. This also explains why *Moldyn* cannot be predicted with both high accuracy and high coverage—the neighbor search has varying behavior since a particle may have a different number of neighbors. The low recall in other programs has the same reason: the automatic analysis is more thorough than the manual analysis.

Four of the test programs are the simulation of grid, mesh and N-body systems in time steps. Ding and Kennedy showed that they benefited from dynamic data packing, which monitored the run-time access pattern and reorganized the data layout multiple times during an execution [Ding and Kennedy, 1999a]. Their technique was automatic except for a programmer-inserted directive, which must be executed once in each time step. This work was started in part to automatically insert the directive. It has achieved this goal: the largest composite phase in these four programs is the time step loop. Therefore, the phase prediction should help to fully automate dynamic data packing, which is shown by several recent studies to improve performance by integer factors for physical, engineering, and biological simulation and sparse matrix solvers [Ding and Kennedy, 1999a; Han and Tseng, 2000a; Mellor-Crummey et al., 2001; Strout et al., 2003].

Summary For programs with consistent phase behavior, the new method gives accurate locality prediction and consequently yields significant benefits for cache resizing and memory remapping. It is more effective at finding long, recurring phases than previous methods based on program code, execution intervals, their combination, and even manual analysis. For programs with varying phase behavior, the profiling step can often reveal the inconsistency. Then the method avoids behavior prediction of inconsistent phases through a flag (as shown by the experiments reported in Table 4.2). Using a small input in a profiling run is enough for locality phase prediction. Therefore, the technique can handle large programs and long executions. For programs such as *GCC* and *Vortex*, where little consistency exists during the same execution, the locality analysis can still recognize phase boundaries but cannot yet make predictions. Predictions based on statistics may be helpful for these programs, which remains to be our future work. In addition, the current analysis considers only temporal locality. The future work will consider spatial locality in conjunction with temporal locality.

4.4 Related Work

This work is a unique combination of program code and data analysis. It builds on past work in these two areas and complements interval-based methods.

Locality phases Early phase analysis, owing to its root in virtual-memory management, was intertwined with locality analysis. In 1976, Batson and Madison defined a phase as a period of execution accessing a subset of program data [Batson and Madison, 1976]. They showed experimentally that a set of Algol-60 programs spent 90% time in major phases. However, they did not predict locality phases. Later studies used time or reuse distance as well as predictors such as Markov models to improve virtual memory management. Recently, Ding and Zhong found predictable patterns in the overall locality but did not consider the phase behavior [Ding and Zhong, 2003]. We are not aware of any trace-based technique that identifies static phases using locality analysis.

Program phases Allen and Cocke pioneered interval analysis to convert program control flow into a hierarchy of regions [Allen and Cocke, 1976]. For scientific programs, most computation and data access are in loop nests. A number of studies showed that the inter-procedural array-section analysis accurately summarizes the program data behavior. The work by Hsu and Kremer used program regions to control processor voltages to save energy. Their region may span loops and functions and is guaranteed to be an atomic unit of execution under all program inputs [Hsu and Kremer, 2003]. For general purpose programs, Balasubramonian et al. [Balasubramonian et al., 2000b], Huang et al. [Huang et al., 2003], and Magklis et al. [Magklis et al., 2003] selected as phases procedures whose number of instructions exceeds a threshold in a profiling run. The three studies found the best voltage for program regions on a training input and then tested the program on another input. They observed that different inputs did not affect the voltage setting. The first two studies also measured the energy saving of phase-based cache resizing [Balasubramonian et al., 2000b; Huang et al., 2003]. A recent work by Lau et al. considers loops, procedures, and call sites as possible phase markers if the variance of their behavior is lower than a relative threshold [Lau et al., 2006]. In comparison, the new technique does not rely on static program structure. It uses trace-based locality analysis to find the phase boundaries, which may occur anywhere and not just at region, loop or procedure boundaries.

Interval phases Interval methods divide an execution into fixed-size windows, classify past intervals using machine or code-based metrics, and predict future intervals using last value, Markov, or table-driven predictors [Dhodapkar and Smith, 2002, 2003; Duesterwald et al., 2003; Sherwood et al., 2003]. The past work used intervals of length from 100 thousand [Balasubramonian et al., 2000b] to 10 million instructions [Sherwood et al., 2003] and executions from 10 milliseconds to 10 seconds [Duesterwald et al., 2003]. Nagpurkar et al. proposed a framework for online phase detection and explored the parameter space [Nagpurkar et al., 2006]. Interval prediction works well if the interval length does not matter, for example, when an execution consists of long steady phases. Otherwise it is difficult to find the best interval length for a given program on a given input. The experimental data in this paper show the inherent limitation of intervals for programs with constantly changing data behavior. Balasubramonian et al. searches for the best interval size at run time [Balasubramonian et al., 2003]. Their method doubles the interval length until the behavior is stable. Let N be the execution length, this new scheme searches $O(logN)$ choices in the space of N candidates. In this work, we locate phases and determine their exact lengths through off-line locality analysis. We show that important classes of programs have consistent phase behavior and the high accuracy and large granularity of phase prediction allow adaptation with a tight worst-performance guarantee. However, not all programs are amenable to the off-line analysis. Interval-based methods do not have this limitation and can exploit the general class of run-time patterns.

4.5 Summary

This chapter presents a general method for predicting hierarchical memory phases in programs with input-dependent but consistent phase behavior. Based on profiling runs, it predicts program executions hundreds of times larger and predicts the length and locality with near perfect accuracy. When used for cache adaptation, it reduces the cache size by 40% without increasing the number of cache misses. When used for memory remapping, it improves program performance by up to 35%. It is more effective at identifying long, recurring phases than previous methods based on program code, execution intervals, and manual analysis. It recognizes programs with inconsistent phase behavior and avoids false predictions. These results suggest that locality phase prediction should benefit modern adaptation techniques for increasing performance, reducing energy, and other improvements to the computer system design.

Scientifically speaking, this work is another attempt to understand the dichotomy between program code and data access and to bridge the division between off-line analysis and on-line prediction. The result embodies and extends the decades-old idea that locality could be part of the missing link.

However, the method is not universal. It relies on the regular repetitions of program phase behavior. Some applications, such as compilers and interpretors and server programs, have strongly input-dependent behavior. An execution of a compiler, for instance, may compile hundreds of different functions with each compilation showing different behavior. The complexity and irregularity pose a special challenge to phase analysis. The next chapter presents an approach to convert that challenge into an opportunity to detect phases in those programs and effectively benefit dynamic memory management.

5 Behavior Phase Analysis through Active Profiling

Utility programs, which perform similar and largely independent operations on a sequence of inputs, include such common applications as compilers, interpreters, and document parsers; databases; and compression and encoding tools. The repetitive behavior of these programs, while often clear to users, has been difficult to capture automatically. This chapter presents an active profiling technique in which controlled inputs to utility programs are used to expose execution phases, which are then marked, automatically, through binary instrumentation, enabling us to exploit phase transitions in production runs with arbitrary inputs. Experiments with five programs from the SPEC benchmark suites show that phase behavior is surprisingly predictable in many (though not all) cases. This predictability can in turn be used for optimized memory management leading to significant performance improvement.

5.1 Introduction

Complex program analysis has evolved from the static analysis of source or machine code to include the dynamic analysis of behavior across all executions of a program. We are particularly interested in patterns of memory reference behavior, because we can use these patterns to improve cache performance, reduce the overhead of garbage collection, or assist memory leak detection.

A principal problem for behavior analysis is dependence on program input. Outside the realm of scientific computing, changes in behavior induced by different inputs can easily hide those aspects of behavior that are uniform across inputs, and might profitably be exploited. Programming environment tools, server applications, user interfaces, databases, and interpreters, for example, use dynamic data and control structures that make it difficult or impossible for current static analysis to predict run-time behavior, or for profile-based analysis to predict behavior on inputs that differ from those used in training runs.

At the same time, many of these programs have repetitive phases that users understand well at an abstract, intuitive level, even if they have never seen the source code. A C compiler, SPEC CPU2000 GCC for example, has a phase in which it compiles a single input function [Henning, 2000]. It runs this function through the traditional tasks of parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling, and then repeats for the following function.

Most of the applications mentioned above, as well as compression and transcoding filters, have repeating *behavior phases*, and often subphases as well. We refer to such programs as *utilities*. They have the common feature that they accept, or can be configured to accept, a sequence of requests, each of which is processed more-or-less independently of the others. Program behavior differs not only across different inputs but also across different parts of the same input, making it difficult for traditional analysis techniques to find the phase structure embodied in the code. In many cases, a phase may span many functions and loops, and different phases may share the same code.

Figure 5.1 illustrates the opportunities behavior phases provide for memory management. Though the volume of live data in the compiler may be very large while compiling an individual, it always drops to a relatively low value at function compilation boundaries. The strong correlation between phases and memory usage cycles

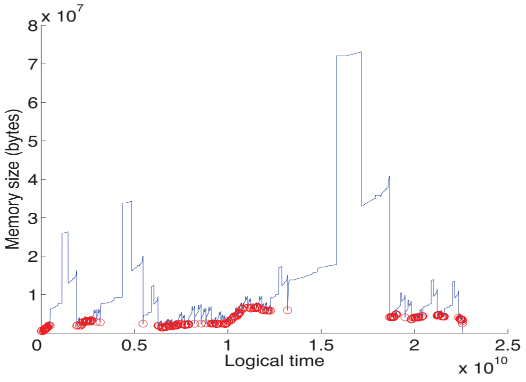
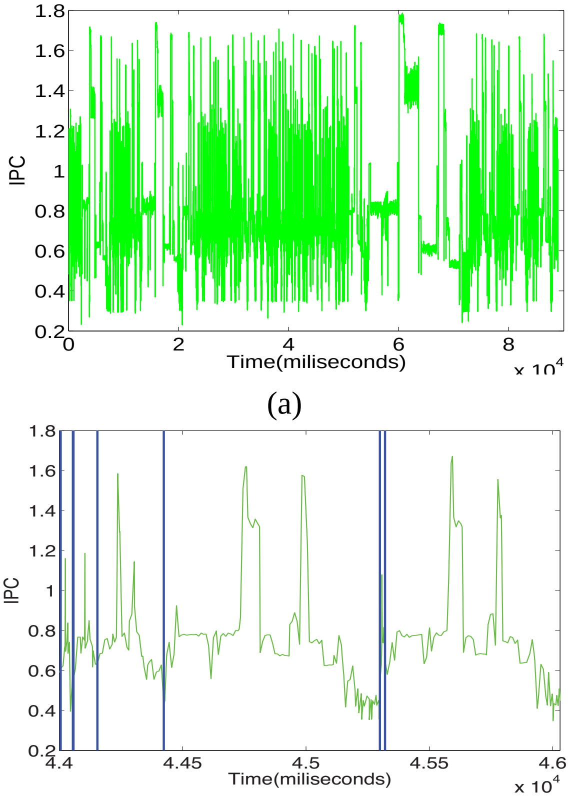


Figure 5.1: The curve of the minimal size of live data during the execution of GCC on input *scilab* with a circle marking the beginning of the compilation of a C function. Logical time is defined as the number of memory accesses performed so far.



(a)

(b)

Figure 5.2: (a) IPC curve of *GCC* on input *scilab* and (b) an enlarged random part. Compilation boundaries are shown as solid vertical lines.

suggests that the phase boundaries are desirable points to reclaim memory, measure space consumption to predict memory usage trends, and classify object lifetimes to assist in memory leak detection.

Figure 5.1, however, also illustrates the challenges in detecting phases. Phases differ greatly in both length and memory usage. Other metrics show similar variation. Figure 5.2(a), for example, plots IPC (instruction per cycle) for the same program run (with physical time on the x axis). The strong dependence of curve shape on the function being compiled makes it difficult for traditional analysis techniques to find the phase structure embodied in the code. A zoomed view of the curve (Figure 5.2(b)) suggests that something predictable is going on: IPC in each instance has two high peaks in the middle and a declining tail. But the width and height of these features differs so much that previous signal-processing based phase analysis, as the locality phase analysis in Chapter 4, cannot reliably identify the pattern [Shen et al., 2004b]. Furthermore, the phases of GCC and other utility programs typically span many function calls and loops, precluding any straightforward form of procedure or loop based phase detection.

In this paper we introduce *active profiling*, which addresses the phase detection problem by exploiting the following observation:

> If we control the input to a utility program, we can often force it to display an artificially regular pattern of behavior that exposes the relationship between phases and fragments of machine code.

Active profiling uses a sequence of identical requests to induce behavior that is both representative of normal usage and sufficiently regular to identify outermost phases (defined in Section 5.2.1). It then uses different real requests to capture inner phases and to verify the representativeness of the constructed input. In programs with a deep phase hierarchy, the analysis can be repeated to find even lower level phases. We can also design inputs to target specific aspects of program behavior, for example, the compilation of loops.

Utility programs are the ideal target for this study because they are widely used and commercially important, and because users naturally understand the relationship between inputs and top-level phases. Our technique, which is fully automated, works on programs in binary form. No knowledge of loop or function structure is required, so a user can apply it to legacy code. Because users control the selection of regular inputs, active profiling can also be used to build specialized versions of utility programs for different purposes, breaking away from the traditional “one-binary-fits-all” program model.

We evaluate our techniques on five utility programs from the SPEC benchmark suites. For each we compare the phases identified by active profiling with phases based on static program structure (functions and loop nests) and on run-time execution intervals. Finally, we demonstrate the use of phase information to monitor memory usage, improve the performance of garbage collection, and detect memory leaks.

5.2 Active Profiling and Phase Detection

5.2.1 Terminology

Program phases have a hierarchical structure. For utility programs, we define an *outermost phase* as the processing of a request, such as the compilation of a function in a C compiler, the compression of a file in a file compressor, and the execution of a query on a database. An *inner phase* is a computation stage in the processing of a request. Compilation, for example, typically proceeds through parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling. A *phase marker* is a basic block that is always executed near the beginning of that phase, and never otherwise.

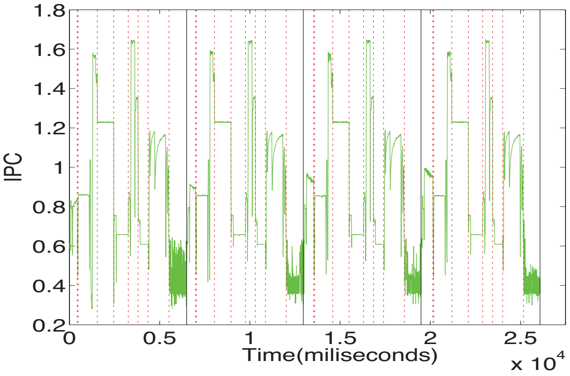


Figure 5.3: IPC curve of *GCC* on an artificial regular input, with top-level (solid vertical lines) and inner-level (broken vertical lines) phase boundaries.

5.2.2 Constructing Regular Inputs

In utility programs, phases have variable length and behavior as shown in Figure 5.2. We can force regularity, however, by issuing a sequence of identical requests—in GCC, by compiling a sequence of identical functions, as shown in Figure 5.3. Solid and broken vertical lines indicate outermost and inner phase boundaries, identified by our analysis. The fact that behavior repeats a predetermined number of times (the number of input repetitions) is critical to the analysis.

A utility program provides an interface through which to make requests. A request consists of data and requested operations. The interface can be viewed as a mini-language. It can be as simple as a stream of bytes and a small number of command-line arguments, as, for example, in a file compression program. It can also be as complicated as a full-fledged programming language, as for example, in a Java interpreter or a simulator used for computer design.

To produce a sequence of repeating requests, we can often just repeat a request if the service is stateless—that is, the processing of a request does not change the internals of the server program. File compression, for example, is uniformly applied to every input file; the compression applied to later files is unaffected by the earlier ones. Care must be taken, however, when the service stores information about requests. A compiler generally requires that all input functions in a file have unique names, so we replicate the same function but give each a different name. A database changes state as a result of insertions and deletions, so we balance insertions and deletions or use inputs containing only lookups.

The appropriate selection of regular inputs is important not only to capture typical program behavior, but also to target analysis at subcomponents of a program. For example, in *GCC*, if we are especially interested in the compilation of loops, we can construct a regular input with repeated functions that have nothing but a sequence of identical loops. Phase detection can then identify the inner phases devoted to loop compilation. By constructing special inputs, not only do we isolate the behavior of a sub-component of a service, we can also link the behavior to the content of a request. We will discuss the use of targeted analysis for a *Perl* interpreter in Section 5.3.2.

5.2.3 Selecting Phase Markers

Active profiling finds phase markers in three steps. The first step searches for regularity in the basic-block trace induced by a regular input; this indicates outermost phases. The second and third steps use real inputs to check for consistency and to identify inner phases.

Using a binary instrumentation tool, we modify the application to generate a dynamic trace of basic blocks. Given a regular input containing f requests, the trace should contain f nearly identical subsequences. The phase markers must be executed f times each, with even intervening spaces.

We first purify the block trace by selecting basic blocks that are executed f times. Not all such blocks represent actual phase boundaries. A block may happen to be executed f times during initialization, finalization, memory allocation, or garbage col-

Data Structure

$innerMarkers$: the set of inner phase markers
$outerMarker$: the outermost phase marker
$traceR$: the basic block trace recorded in the regular training run
$traceI$: the basic block trace recorded in the normal (irregular) training run
$RQSTR$: the number of requests in the regular input
$RQSTI$: the number of requests in the normal input
$setB$: the set of all basic blocks in the program
$setB1, setB2, setB3$: three initialy empty sets
b_i : a basic block in $setB$
$timeR(b_i, j)$: the instructions executed so far when b_i is accessed for the jth time
$V_i =< {V_i}^1, {V_i}^2, \ldots, {V_i}^k >$: the recurring distance vector of basic block b_i in $traceR$, where ${V_i}^j = timeR(b_i, j+1) - timeR(b_i, j)$

Algorithm

step 1) Select basic blocks that appear $RQSTR$ times in $traceR$ and put them into $setB1$.
step 2a) From $setB1$, select basic blocks whose recurring distance pattern is similar to the majority and put them into $setB2$.
step 2b) From $setB2$, select basic blocks that appear $RQSTI$ times in $traceI$ and put them into $setB3$.
step 3) From $setB3$, select basic blocks that are followed by a long computation in $traceR$ before reaching any block in $setB3$ and put those blocks into $innerMarkers$; $outerMarker$ is the block in $innerMarkers$ that first appears in $traceR$.

```
Procedure Step2a()
// $M$ and $D$ are two initially empty arrays
  for   every $b_i$ in $setB1$ {
        $V_i$ = GetRecurDistance($b_i$, $traceR$);
        $m_i$ = GetMean($V_i$);
        $d_i$ = GetStandardDeviation($V_i$);
        $M$.Insert($m_i$);
        $D$.Insert($d_i$);}
  if    (!IsOutlier($m_i$,$M$) &&
        !IsOutlier($d_i$,$D$)){
        $setB2$.AddMember($b_i$);}
End
```

```
Procedure IsOutlier($x$, $S$)
// $S$ is a container of values
  $m$ = GetMean($S$);
  $d$ = GetStandardDeviation($S$);
  if ($|x - m| > 3 * d$) return true;
  return false;
End
```

Figure 5.4: Algorithm of phase marker selection and procedures for recurring-distance filtering.

lection. We therefore measure the mean and standard deviation of distance between occurrences, and discard blocks whose values are outliers (see Figure 5.4).

The remaining code blocks all have f evenly spaced occurrences, but still some may not be phase markers. In *GCC*, for example, the regular input may contain a single branch statement. Code to parse a branch may thus occur once per request with this input, but not with other inputs. In step two we check whether a block occurs consistently in other inputs. We use a real input containing g (non-identical) requests. We measure the execution frequency of the candidate blocks and keep only those that are executed g times. Usually one real input is enough to remove all false positives, but this step can be repeated an arbitrary number of times to increase confidence.

Having identified blocks that appear always to occur exactly once per outermost phase, we consider the possibility that these may actually mark interesting points *within* an outermost phase. Compilation, for example, typically proceeds through parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling. We call these *inner phases*. Each is likely to begin with one of the identified blocks.

In step three we select inner phases of a non-trivial length and pick one block for each phase boundary. Figure 5.5 shows a trace of *GCC* on regular input. Each circle on the graph represents an instance of a candidate inner phase marker. The x-axis represents logical time (number of memory accesses); the y-axis shows the identifier (serial number) of the executed block. We calculate the logical time between every two consecutive circles: the horizontal gaps in Figure 5.5. From these we select the gaps whose width is more than 3 standard deviations larger than the mean. We then designate the basic block that precedes each such gap to be an inner-phase boundary marker. The first such marker doubles as the marker for the outermost phase.

The phases of a utility program may also nest. For example, the body of a function in the input to a compiler may contain nested statements at multiple levels. This nesting may give rise to deeply nested phases, which our framework can be extended to identify, using a sequence of identical sub-structures in the input. In the case of the compiler, we

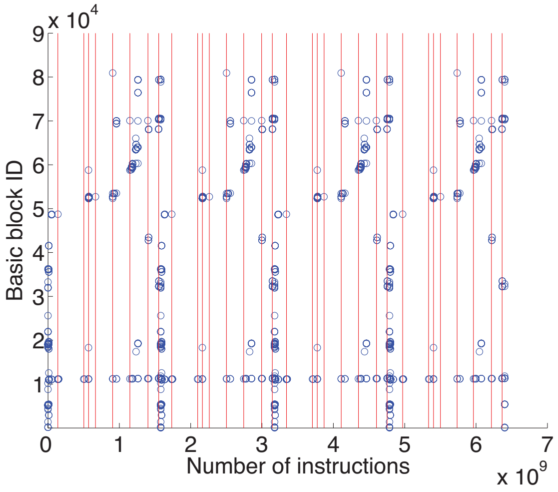


Figure 5.5: *GCC* inner-phase candidates with inner-phase boundaries.

can construct a function with a sequence of identical loop statements, and then mark the portions of each inner phase (compilation stage) devoted to individual loops, using the same process that we used to identify outermost phases in the original step of the analysis.

5.3 Evaluation

We test six programs, shown in Table 5.1, from the SPEC95 and SPEC2K benchmark suites: a file compression utility, a compiler, two interpreters, a natural language parser, and an object-oriented database. Three other utility programs—two more compression utilities—exist in these two suites. We have not yet experimented with them because they do not contribute a new application type. All test programs are written in C. Phase analysis is applied to the binary code.

We construct regular inputs as follows. For *GCC* we use a file containing 4 identical functions, each with the same long sequence of loops. For *Compress*, which is written to compress and decompresses the same input 25 times, we provide a file that is 1% of the size of the reference input in the benchmark suite. For *LI*, we provide 6 identical

Table 5.1: Benchmarks for utility phase analysis

Benchmark	Description	Source
Compress	UNIX compression utility	SPEC95Int
GCC	GNU C compiler 2.5.3	SPEC2KInt
LI	Xlisp interpreter	SPEC95Int
Parser	natural language parser	SPEC2KInt
Vortex	object oriented database	SPEC2KInt
Perl	Perl interpreter	SPEC2KInt

expressions, each of which contains 34945 identical sub-expressions. For *Parser* we provide 6 copies of the sentence “John is more likely that Joe died than it is that Fred died.” (That admittedly nonsensical sentence is drawn from the reference input, and not surprisingly takes an unusually long time to parse.) The regular input for *Vortex* is a database and three iterations of lookups. Since the input is part of the program, we modify the code so that it performs only lookups but neither insertions nor deletions in each iteration.

We use ATOM [Srivastava and Eustace, 1994] to instrument programs for the phase analysis on a decade-old Digital Alpha machine, but measure program behavior on a modern IBM POWER4 machine through its Due to the lack of a binary rewriting tool on the IBM machine, we insert phase markers into the Alpha binary, manually identify their location, insert the same markers at the source level, and then compile and run the marked program on the IBM platform. hardware performance monitoring facilities. POWER4 machines have a set of hardware counters, which are automatically read every 10ms. The AIX 5.1 operating system provides a programming interface library called PMAPI to access those counters. By instrumenting the program with the library function calls, one can determine the set of the hardware events specified by the user at the instrumentation point. The instrumentation is also set to automatically generate an interrupt every 10ms so that the hardware counters are read at the 10ms granularity. Not all hardware events can be measured simultaneously. We collect cache miss rates and IPCs (in a single run) at the boundaries of program phases and, within phases, at 10ms intervals.

The phase detection technique finds phases for all 6 benchmarks. *GCC* is the most complex program and shows the most interesting behavior. *Perl* has more than one type of phase. We describe these in the next two subsections, and the remaining programs in the third subsection.

5.3.1 GCC

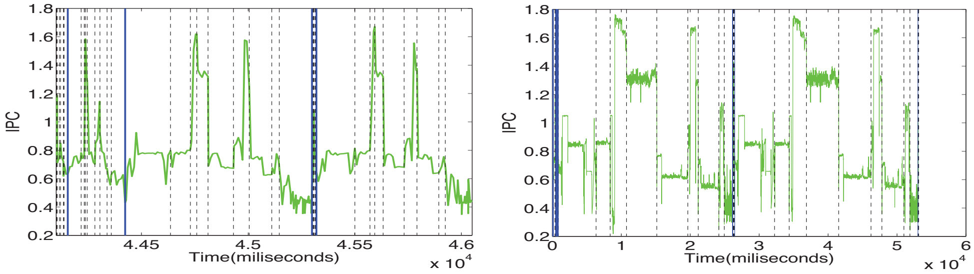
GCC comprises 120 files and 222182 lines of C code. The phase detection technique successfully finds the outermost phase, which begins the compilation of an input function. We also find 8 inner phases. Though the analysis tool never considers the source, we can, out of curiosity, map the automatically inserted markers back to the source code, where we discover that the 8 markers separate different compilation stages.

The first marker is at the end of function “loop_optimize”, which performs loop optimization on the current function. The second marker is in the middle point of function “rest_of_compilation”, where the second pass of common sub-expression elimination completes. The third and fourth markers are both in function “life_analysis”, which determines the set of live registers at the start of each basic block and propagates the life information inside the basic block. The two markers are separated by an analysis pass, which examines each basic block, deletes dead stores, generates auto-increment addressing, and records the frequency at which a register is defined, used, and redefined. The fifth marker is in function “schedule_insns”, which schedules instructions block by block. The sixth marker is at the end of function “global_alloc”, which allocates pseudo-registers. The seventh marker is in the same function as the fifth marker, “schedule_insns”. However, the two markers are in different branches, and each invocation triggers one sub-phase but not the other. The two sub-phases are executed through two calls to this function (only two calls per compilation of a function), separated by the sixth marker in “global_alloc” among other function calls. The last marker is in the middle of function “dbr_schedule”, which places instructions into delay slots. These automatically detected markers separate the compilation into 8 major stages. Given the complexity of the code, manual phase marking would be extremely difficult for someone who does not know the program well. Even for an expert in *GCC*, it might not be easy to identify sub-phases that occupy large portions of the execution time, of roughly equal magnitude.

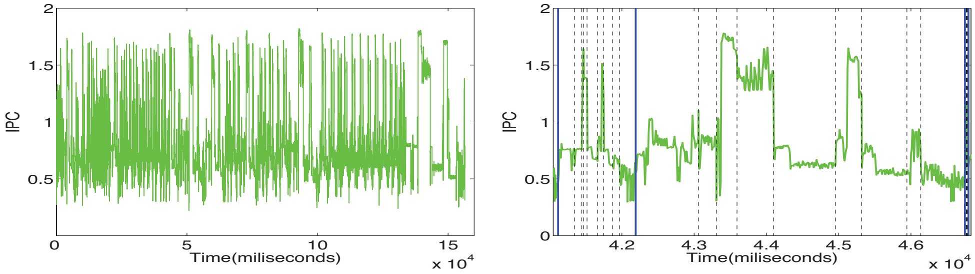
GCC behavior varies with its input. Regularity emerges, however, when we cut the execution into phases. Figure 5.6(a) shows the same curve as Figure 5.2(b) with markings for outermost (solid) and inner (broken) phases. Both outermost and inner phases show similar signal curves across phase instances. The IPC curves of GCC on other inputs have a related shape, shown in Figure 5.6(b)–(d). This shows that *GCC* displays a recurring execution pattern—the same complex compilation stages are performed on each function in each input file. The outermost phase and inner phase markers accurately capture the variation and repetition of program behavior, even when the shape of the curve is not exactly identical from function to function or from input to input. Note that while we have used IPC to illustrate behavior repetition, the phase marking itself is performed off-line and requires no on-line instrumentation.

The lower four graphs in Figure 5.6 show the IPC curves of *Compress*, *Vortex*, *LI*, and *Parser*. We will discuss them in Section 5.3.3 when comparing behavior phases from active profiling with other types of phases.

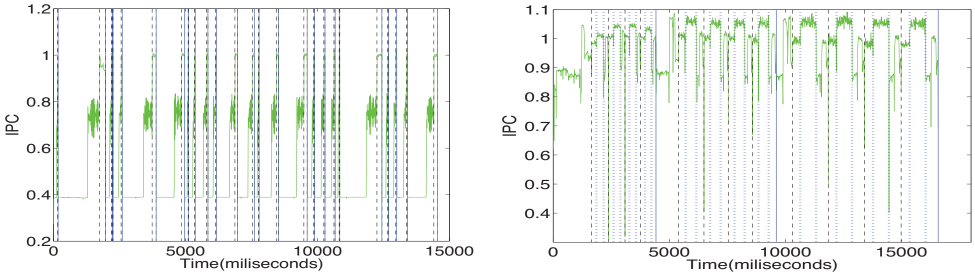
Figure 5.8(a) shows a distribution graph of IPC and cache hit rates for phase instances of *GCC*. Instances of different sub-phases are represented by different symbols. *GCC* has 57 instances of the outermost phase in that ref input. Each instance is divided into 8 inner phases. We have a total of 456 points in Figure 5.8(a). The 456 points cluster into 5 rough groups. The top group is the cluster of phase 3. It corresponds to the highest peak in the IPC curve, and is separated from the other phase instances.



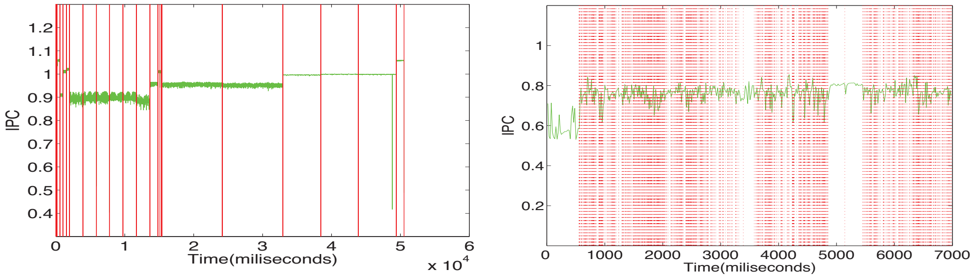
(a) GCC on *scilab* enlarged (b) GCC on *166*



(c) GCC on *200* without phase markers (d) GCC on *200* enlarged



(e) Compress (f) Vortex



(g) Li (h) Parser

Figure 5.6: IPC curves of GCC, Compress, Vortex, Li and Parser with phase markers

The cluster of phase 4 overlaps with the top of the cluster of phase 6, but separates from its major body. Phase 4 corresponds to the highland in the IPC curve, and phase 5 corresponds to the second highest peak with some low transition parts. The cluster of phase 8 corresponds to a short segment in IPC curve with low IPC and high cache hit rate. It separates from the other clusters well. The remaining large cluster contains the instances of phases 1, 2, 5 and 7. These four phases correspond to the 4 lowest IPC segments. They are close to each other but still separate mostly. Phase 2 has the highest cluster, phase 7 the lowest, with phase 1 in the middle. Phase 5 has the rightmost cluster with highest cache hit rate. Most of the 8 groups are very tight except for the values from phase 2. Even for this group, most points have almost the same IPC, and cache hit rates that vary by less than 0.2.

5.3.2 Perl

Though our active analysis tool is usually employed in a fully automated form (the user provides a regular input and a few real inputs, and the tool comes back with an instrumented binary), we can invoke the sub-tasks individually to explore specific aspects of an application.

As an example, consider the *Perl* interpreter. The installed version in our system directory has 27 thousand basic blocks and has been stripped of all debugging information. Perl interprets one program at a time, so it does not have outermost phases as other programs do. In hopes of exploring how the interpreter processes function calls, however, we created a regular 30-line input containing 10 identical calls. Given this input, the regularity checking tool (step 1 of Section 5.2.3) identified 296 candidate marker blocks. We then created a 10-line irregular program containing three calls to two different functions. The consistency checking tool (step 2) subsequently found that 78 of the 296 candidates appeared consistently. Choosing one of these blocks at random (number 5410, specifically), we tested a third input, written to recursively sum the numbers from 1 to 10 in 11 calls. Block 5410 was executed exactly 11 times. This experience illustrates the power of active profiling to identify high-level patterns in low-level code, even when subsumed within extraneous computation.

5.3.3 Comparison with Procedure and Interval Phase Analysis

In this section, we compare the ability of different analysis techniques—active profiling, procedure analysis, and interval analysis—to identify phases with similar behavior. In Section 5.4 we will consider how to use these phases to optimize memory management. Different metrics—and thus different analysis techniques—may be appropriate for other forms of optimization (e.g., fine-grain tuning of dynamically configurable processors).

Program phase analysis takes a loop, subroutine, or other code structures as a phase [Balasubramonian et al., 2000b; Georges et al., 2004; Huang et al., 2003; Lau et al., 2004; Liu and Huang, 2004; Magklis et al., 2003]. For this experiment, we mainly consider procedure phases and follow the scheme given by Huang et al., who picked subroutines by two thresholds, θ_{weight} and θ_{grain} [Huang et al., 2003]. Assume the execution length is N. Their scheme picks a subroutine p if the cumulative time spent in p (including its callees) is no less than $\theta_{weight}T$ and the average time per invocation no less than $\theta_{grain}T$. In other words, the subroutine is significant and does not incur an excessive overhead. Huang et al. used 5% for θ_{weight} and 10K instructions for $\theta_{grain}T$. Georges et al. made the threshold selection adaptive based on individual programs, the tolerable overhead, and the need of a user [Georges et al., 2004]. They studied the behavior variation of the procedure phases for a set of Java programs. Lau et al. considered loops and call sites in addition to subroutines and selected phases whose behavior variation is relatively low [Lau et al., 2004]. In this experiment, we use the fixed thresholds from Huang et al. The extension by Lau et al. may reduce the behavior variation seen by in our experiments.

Profile-based phase analysis depends on the choice of the training input. In our test set, the number of outermost phase instances range from 3 queries in *Vortex* to 850 sentences in *Parser*. The threshold θ_{grain} would need to less than 0.13%. Many procedures may qualify under a small θ_{grain}, making it difficult to find the procedure for the outermost phase, if such procedure exists. In addition, the behavior of phase instances of a utility program may differ significantly. For example in *Li*, the IPC of the outermost phase has an unpredictable pattern, so consistency-based selection may find only the whole program as a phase. Indeed, the execution speed may vary greatly in *Li* as the interpreter processes different Lisp expressions. As we will show later, such phase is still valuable for memory management because each phase instance, regardless of its IPC or cache miss rate variation, represents a memory usage cycle. Active profiling, as guided by a user, does not depend on the same empirical thresholds. It considers all program instructions as possible phase boundaries not just procedures and other code regions. In addition, active profiling uses much smaller inputs. For example, the regular input to *Parser* contains only 6 sentences.

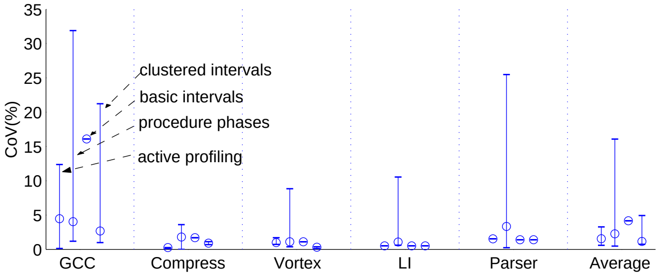
Interval analysis divides an execution into fixed-size windows, classifies past intervals using machine or code-based metrics, and predicts the class of future intervals using last value, Markov, or table-driven predictors [Balasubramonian et al., 2000b; Dhodapkar and Smith, 2002; Duesterwald et al., 2003; Sherwood et al., 2003]. Most though not all past studies use a fixed interval length for all executions of all programs, for example, 10 million or 100 million instructions. For purposes of comparison, we select the interval length for each program in our experiments so that the total number of intervals equals the number of inner behavior phase instances identified by active profiling. Space limitations do not permit us to consider all possible prediction and clustering methods. We calculate the upper bound of all possible methods using this interval length by applying optimal partitioning (approximated by k means in practice) on the intervals of an execution. We further assume perfect prediction at run-time—we assume knowledge of the number of clusters, the behavior, and the cluster membership of each interval before execution.

Though phases are not in general expected to have uniform internal behavior, different instances of the same phase should have similar *average* behavior. In our experiments we consider cache hit rate and IPC as measures of behavior. Quantitatively, we compute the *coefficient of variation (CoV)* among phase instances, which is the standard deviation divided by the mean. The CoV is the expected difference between the prediction (the average) and the actual value of each phase. In a normal distribution, a standard deviation of d means that 68% of the values fall in the range $[m-d, m+d]$ and 95% fall in the range $[m-2d, m+2d]$. The results from our hardware counters are not accurate for execution lengths shorter than 10ms, so we excluded phase instances whose lengths are shorter than 10ms.

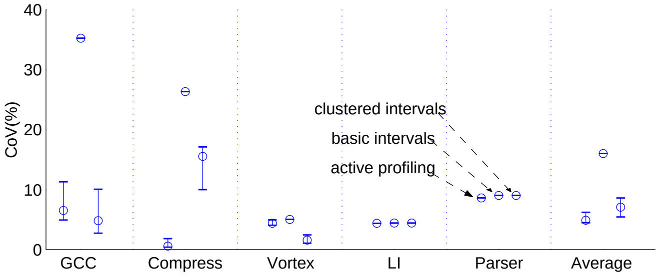
Figure 5.7(a) shows the CoVs of cache hit rates. Each program is shown by a group of floating bars. Each bar shows the CoV of a phase analysis method. When a program has multiple inner phases, the two end points of a bar show the maximum and minimum and the circle shows the average. The four bars in each group show the CoVs of behavior phases, procedure phases, intervals with no clustering (all intervals belong to the same group), and intervals with k-means clustering (the best possible prediction given the number of clusters).

Unlike the other methods, the results for procedure phases are obtained via simulation. Since some of the procedures are library routines, we would require binary instrumentation to obtain equivalent results from hardware counters. We use simulation because we lack an appropriate tool for the IBM machine.

GCC has 8 behavior sub-phases. The CoV is between 0.13% and 12%, and the average is 4.5%. The CoV for procedure phases ranges from 1.2% to 32% with an average of 4%. When cutting the execution into the same number of fixed length intervals as the number of inner phase instances, the CoV is 16%. When the intervals are clustered into 8 groups, the CoV ranges from 1% to 22% with an average of 2.7%.



(a) Cache hit rate CoV



(b) IPC CoV

Figure 5.7: Behavior consistency of four types of phases, calculated as the coefficient of variance among instances of each phase. For each program, the range of CoV across all inner phases is shown by a floating bar where the two end points are maximum and minimum and the circle is the average. A lower CoV and a smaller range mean more consistent behavior. Part (b) shows the CoV of IPC.

The average CoV for procedure phases and interval phases is lower than that of the behavior phases. However, the procedure phases do not cover the entire execution, and the interval results assume perfect clustering and prediction. In addition, the behavior phase that has the highest consistency (0.13% CoV) is the 4th subphase, which represents 8% of the program execution. The boundaries of this sub-phase are not procedure boundaries. The phase length varies rather than staying constant. As a result, neither procedure nor interval analysis, under however ideal circumstances, could identify and predict this behavior.

Compress has two sub-phases. The cache hit rate is always 88% for instances of the first sub-phase and 90% for those of the second sub-phase, despite the fact that the instances have different lengths, as shown in Figure 5.6(e). The relative length ratio is constant. In each outermost phase, the first sub-phase takes 88% of the time and the second takes the remaining 12%. The CoVs of the two sub-phases are 0.15% and

0.21%, barely visible in Figure 5.7 (a). When divided into two clusters, the smallest and average CoV from interval phases is 0.7% and 0.9%. This program shows the value of variable-length phases: even the ideal clustering of fixed length intervals cannot be as accurate.

Vortex has less behavior variation than the previous two programs. The best case procedure and interval phase results are 0.3% CoV, better than the 0.7% minimum CoV of behavior phases. The highest CoV, 8.9%, occurs in a procedure phase. For predicting the cache hit rate, the behavior phase information is not very critical. A carefully picked interval length may capture a similar stable behavior. However, behavior phases still have the advantage of not needing to pick an interval length.

LI shows very few performance changes, as seen in Figure 5.6(g). Except for procedure phases, all methods have a CoV of less than 1%. The worst procedure, however, shows an 11% CoV. *Parser* is similar. The CoV is below 2% except for procedure phases, which have a CoV of 3% on average and 26% in the worst case. The two programs show that the behavior variation for a procedure can be large even for a program with relatively constant overall behavior. The results also show the difficulty of setting thresholds in procedure and interval phase analysis. A CoV of 1% may be too large for *LI* but too small for programs such as *GCC*.

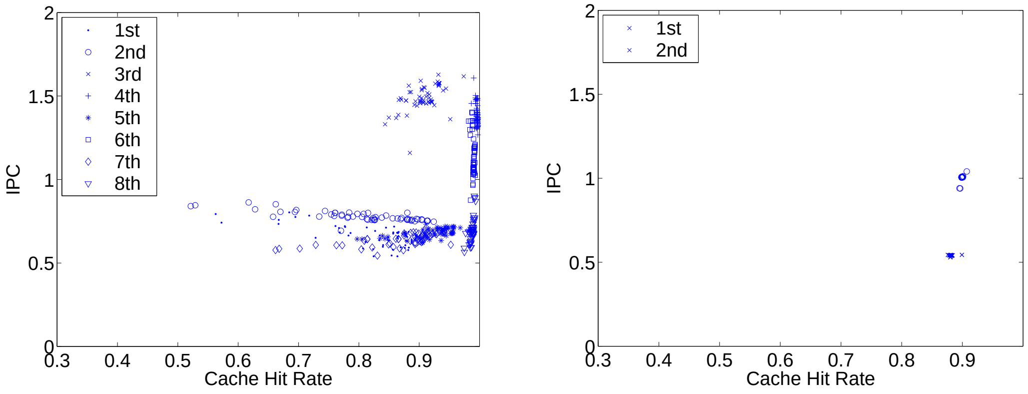
The CoVs of the programs' IPC are shown in Figure 5.7(b). We do not include the procedure-based method for IPC since it is based on simulation and therefore could not be directly compared to the real measurement of IPCs in the other three cases. Between the behavior and interval phases, the qualitative results are the same for IPC as for the cache hit rate. On average across all programs, the CoV is 4.9% for behavior phases and 7.1% for intervals with optimal clustering and prediction.

The five programs show a range of behavior. *Compress* is at one extreme, with behavior that is highly varied but consistent within sub-phases. *LI* is at the other extreme, with behavior that is mostly constant and that does not change between phases. Graphically, the two graphs in Figure 5.8 plot the cache hit rate and IPC on a two-dimensional plot. Each phase instance is a point. In the first graph for *Compress*, the points are in two tight clusters. In the second graph for *LI*, the points spread within an range. The behavior variation of the other three programs is between these two extremes. Note that the interval phases do not produce highly clustered points as in Figure 5.8(a). These points have variable lengths, which currently can only be marked by behavior phases.

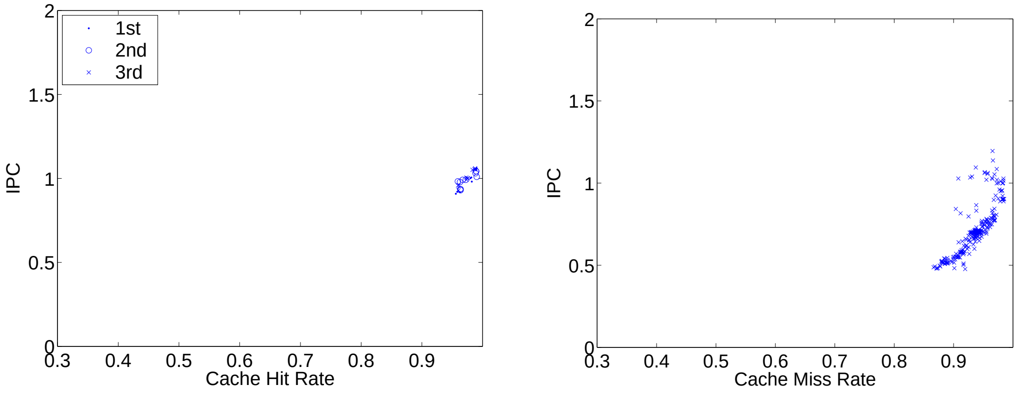
Recently, Georges et al. [Georges et al., 2004] and Lau et al. [Lau et al., 2004] improved procedure phase analysis by picking procedures whose behavior variation is below a threshold. Their method can avoid procedures with a high CoV. The best possible result is shown by the lower bound CoV in Figure 5.7. However, procedure phases, especially those with consistent behavior, may not cover the entire execution. Setting the threshold for each program is not trivial. Procedure phases cannot capture behavior phases that do not start or end on procedure or loop boundaries, for example, the 4th sub-phase of *GCC* as discussed above. Finally, memory phases are valuable for garbage collection (as we show next), even though their instances have highly varied CPU or cache behavior.

5.4 Uses of Behavior Phases

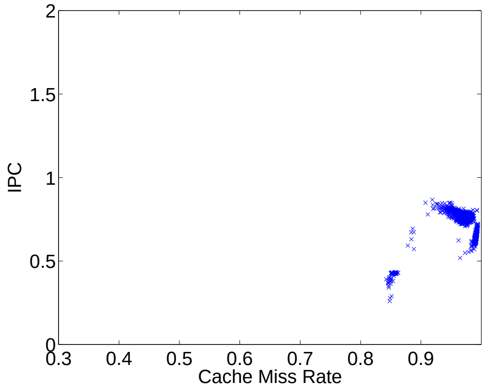
Behavior phases allow a programmer to improve the performance of commonly used programs on conventional hardware. Program dynamic data allocation sites can be classified into "phase local" and "global" according to the living period of their allocated data. It in turn helps the detection of memory leaks: if there are objects from "phase local" sites that are not released after a phase instance, the allocation could be memory leak. We also applied phase information to preventive memory management: Garbage collection is invoked at phase boundaries only except a hard memory bound is reached in a phase. The experiment on program *LI* shows 44% speedup compared to the original program [Ding et al., 2005]. For Java programs, the phase information



(a) GCC: 456 instances of 8 phases (b) Compress: 50 instances of 2 phases



(c) Vortex: 36 instances of 3 phases (d) Li: 271 instances of 1 phase



(e) Parser: 703 instances of 1 phase

Figure 5.8: IPC and cache hit rate distribution graphs.

enables program-level memory control: We use an adaptive scheme to explore at phase boundaries to find the good heap size [Zhang et al., 2006].

5.5 Related Work

Early phase analysis was aimed at virtual memory management and was intertwined with locality analysis. In 1976, Batson and Madison defined a phase as a period of execution accessing a subset of program data [Batson and Madison, 1976]. Later studies used time or reuse distance as well as predictors such as Markov models to improve virtual memory management. Recently, Shen et al. used reuse distance to model program behavior as a signal, applied wavelet filtering, and marked recurring phases in programs [Shen et al., 2004b]. For this technique to work, the programs must exhibit repeating behavior. By using active profiling, we are able to target utility programs, whose locality and phase length are typically input-dependent, and therefore not regular or uniform.

Balasubramonian et al. [Balasubramonian et al., 2000b], Huang et al. [Huang et al., 2003; Liu and Huang, 2004], and Magklis et al. [Magklis et al., 2003] selected as program phases procedures, loops, and code blocks whose number of instructions exceeds a given threshold during execution. For Java programs, Georges et al. selected as phases those procedures that display low variance in execution time or cache miss rate [Georges et al., 2004]. Lau et al. considered loops, procedures, and call sites as possible phase markers if the variance of their behavior is lower than a relative threshold [Lau et al., 2004]. It is not easy to determine the expected size or behavior variance for phases of a utility program when one has no control over the input. For example, instances of the compilation phase may have very different execution length and memory usage.

Allen and Cocke pioneered interval analysis to model a program as a hierarchy of regions [Allen and Cocke, 1976]. Hsu and Kremer used program regions to control processor voltages to save energy. Their regions may span loops and functions and are guaranteed to be an atomic unit of execution under all program inputs [Hsu and Kremer, 2003].

In comparison to program phases, active profiling does not rely on the static program structure. It considers all program statements as possible phase boundaries. We found that in *GCC*, some sub-phase boundaries were methods called inside one branch of a conditional statement. In addition, active profiling relies on a user to target specific behavior rather than on empirical thresholds that may need to be tuned for each target machine or class of input.

Interval methods divide an execution into fixed-size windows, classify past intervals using machine or code-based metrics, and predict the behavior of future intervals using last value, Markov, or table-driven predictors (e.g., [Balasubramonian et al., 2000b; Dhodapkar and Smith, 2002; Duesterwald et al., 2003; Sherwood et al., 2003]). Balasubramonian et al [Balasubramonian et al., 2003] dynamically adjust the size of the interval based on behavior predictability/sensitivity. However, since the intervals don't match phase boundaries, the result may be an averaging of behavior across several phases. Duesterwald et al. gave a classification of these schemes [Duesterwald et al., 2003]. Nagpurkar et al. proposed a framework for online phase detection and explored the parameter space [Nagpurkar et al., 2006]. A fixed interval may not match the phase length in all programs under all inputs. Our technique finds variable-length phases in utility programs. It targets program level transformations such as memory management and parallelization, so it is designed for different purposes than interval phase analysis is.

5.6 Future Directions

Locality phases and behavior phases can successfully detect and predict large-scale phase behavior without any application-dependent threshold. However, both techniques require one or more profiling runs with complicated analysis, and either of them is only applicable to a class of programs. Those limitations may be tolerable to the uses on some important applications. But for general users, it is desirable to have a universal approach with lightweight analysis but without the limitations of current approaches.

The second extension exists in the connection between program concurrency and phases. Computers are increasingly equipped with multi-level parallelism: instruction-level, thread-level on a single chip, thread-level across chips, and parallelism across machines. Even on the same level, the granularity could vary a lot. A critical problem of exploiting the parallelism is to recognize and predict the concurrency granularity in each phase of a program. TRIPS system [Burger et al., 2004], for example, is a research architecture having more than 32 computing units on a single tile. The TRIPS compiler sends executable code to the hardware in blocks of up to 128 instructions. The processor executes a block all at once, as if it were a single instruction, greatly decreasing the overhead associated with instruction handling and scheduling. Instructions inside a block execute in a "data flow" fashion, meaning that each instruction executes as soon as its inputs arrive, rather than in some sequence imposed by the compiler or the programmer. Phase analysis could be extended to predict the level of concurrency and guide the parallel execution. Another example is contention-aware and resource-aware automatic parallelization. With the knowledge of the contention and available resource in the future, the programming system can dynamically decide the number of threads and the granularity.

5.7 Summary

The chapter has presented active profiling for phase analysis in utility programs, such as compilers, interpreters, compression and encoding tools, databases, and document parsers. By reliably marking large-scale program phases, active profiling enables the implementation of promising new program improvement techniques, including preventive garbage collection (resulting in improved performance relative to standard reactive collection), memory-usage monitoring, and memory leak detection.

Using deliberately regular inputs, active profiling exposes top-level phases, which are then marked via binary instrumentation and verified with irregular inputs. The technique requires no access to source code, no special hardware support, no user knowledge of internal application structure, and no user intervention beyond the selection of inputs. The entire process is fully automated, from the scripting of profiling runs, through the collection and analysis of the resulting statistics, to the instrumentation of the program binary to mark application phases and perform the garbage collection or memory monitoring.

Beyond the realm of memory management, we have used active profiling to speculatively execute the phases of utility programs in parallel, obtaining nontrivial speedups from legacy code, which is described in the next chapter.

6 Behavior-Oriented Parallelization

Many programs exhibit some high-level parallelism but may not be parallelizable because of the program's size and complexity. In addition, their parallelism may be dynamic and available with certain inputs only. In this paper we present the design and implementation of *speculative co-processing*, which uses an extra processor to speculatively execute likely parallel regions of code. We show that this speculation improves performance when the parallelism exists. When there is not parallelism, the program still runs correctly and completes as quickly as the unmodified sequential version. We show our software-based implementation of speculative co-processing successfully parallelizes two large open-source applications and a scientific library and improves their performance by up to 86% on a dual-processor PC. The results demonstrate the importance of the three novel features of speculative co-processing: language support for specifying possible (rather than definitive) program behavior, strong isolation to ensure correctness, and using redundant computation to hide the overhead of run-time monitoring and correctness checking.

6.1 Introduction

Most existing programs are written for a sequential machine, yet they often have parallelism that a user understands at the high level, such as, a compression tool that processes data buffer by buffer and file1 by file, an English parser parsing sentence by sentence, and a compiler compiling function by function. Specifying exact parallelism is difficult, however, because thousands of lines of code may be executed at each processing step. Moreover, this parallelism is often dynamic and input dependent.

As modern PCs and workstations are increasingly equipped with multiple processors, it becomes possible and desirable for commonly used legacy software to make use of more than one processor. The problem differs from traditional parallel processing because the target application is large, the parallelism is likely but not guaranteed, and there are often extra processors and memory space that would otherwise be idle if not used. In this work we present *speculative co-processing*, which uses an extra processor and additional memory to speculatively execute likely parallel code regions simultaneously and thereby improve performance over sequential programs.

In general, speculation-based parallelization techniques need to solve three problems: selecting what to speculate, checking for correctness, and recovering from incorrect speculation. We discuss each one in turn. For software implementations, the granularity of speculation needs to be large to amortize the cost of parallel execution on modern processors, which are optimized for uninterrupted, sequential execution. Existing programs are often highly optimized for sequential execution and contain implicit dependences from error handling, buffer reuse, and custom memory management. Although a user can often identify large possibly parallel tasks in a program, the programmer may not know whether dependences exist between tasks let alone the source of the dependence.

Speculative co-processing gives a programmer the ability to indicate *possibly parallel regions (PPR)* in a program by marking the beginning and end of the region with matching markers: *BeginPPR(p)*and *EndPPR(p)*. Figure 6.1 shows an example of the marking of possible loop parallelism, where the loading of the tasks is sequential but the processing of each task is possibly parallel. Figure 6.2 shows the marking of possible function parallelism, where the two calls to a function are possibly parallel. The

```
while (1) {
 get_work();
 ...
 BeginPPR(1);
 step1();
 step2();
 EndPPR(1);
 ...
}
```

Figure 6.1: Possible loop parallelism

```
...
BeginPPR(1);
work(x);
EndPPR(1);
...
BeginPPR(2);
work(y);
EndPPR(2);
...
```

Figure 6.2: Possible function parallelism

semantics of the PPR markers is that *when* BeginPPR(p) *is executed, it is* advisable *to start a (speculative) parallel execution from* EndPPR(p). The markers indicate that the parallelism is likely but not definite, so the parallel execution may have to be canceled and their effect reverted. Such markers may also be inserted by profiling analysis, which can identify frequent but not all possible behavior of a program.

The second problem of speculation is checking for correctness, which entails finding a proof that no dependence in the sequential execution is violated in the reordered execution. In co-processing, we call the non-speculative computation *main* and the speculative ones *spec*. Existing methods of speculation can be differentiated by how the two groups communicate. The first, used by most methods of speculation as well as supporting transactional memory (see Section 6.4), is what we call *weak isolation*, where updates from non-speculative computation become immediately visible to speculative computations. As we explain in Section 6.2.2, this requires fine-grained communication costly to support in software and vulnerable to subtle concurrency errors. Co-processing uses *strong isolation*, where the updates of *main* are made available only at the end. While strong isolation does not support implicit pipelined computation (for which a user can specify explicitly using PPR directives), it enables efficient speculation and rollback, its value based (in addition to dependence based) checking allows certain types of parallel execution in the presence flow dependences, and it allows the program to make use of unmodified hardware and compilers that were designed and optimized for sequential execution. On top of the strong safety net, we include simple improvements to allow early termination of *spec* as soon a conflict is detected.

The last major problem of speculation is safe recovery when speculations fail. In real applications, the speculative work may execute low-level, highly optimized code and perform billions of operations on millions of bytes. It may misbehave in arbitrary ways if it is started on an incorrect state. Unlike most existing methods that use threads, co-processing uses Unix processes for *main* and *spec*, so the entire speculative computation can be terminated without any side effect on the *main's* memory or processor state. Memory copy-on-write has the effect of incremental check pointing and on-demand privatization. The cost of process creation and termination is moderated by the other two features of co-processing: the large granularity of PPR amortizes the cost, and strong isolation removes the need of most inter-process communication.

One unique problem of strong isolation is that the success of *spec* is not known until both *main* and *spec* finish. We present a novel solution, which uses redundant computation through a third process we call the *understudy* to provide an important performance guarantee: *the speculation result is used only when it is correct and when the overheads of speculation—starting, checking, committing—do not outweigh the benefit of the speculation.* This guarantee is achieved through a coordinated ensemble of these processes, which we describe in detail in Section 6.2.3.

Co-processing differs from traditional parallel processing because its goal is not scalable parallel performance but using extra processors for small-scale performance improvement with minimal programming effort by the user. As a parallel programming system, it requires little or no manual changes to the sequential program and no parallel programming or debugging. Furthermore, the system records the causes for the speculation failure, so a programmer can incrementally remove hidden dependences. The programmer can also specialize a program for parallel processing on frequent rather than all inputs. This ease of programming is key to the scalability of a different sort—co-processing for large, existing software.

Co-processing has several limitations. Speculation cannot handle general forms of I/O or other operations with unrecoverable side effects. The current implementation allows limited dynamic allocation within the parallel region and uses only one extra processor for co-processing. The software implementation is best for loosely coupled coarse-grain parallelism but not efficient enough for managing fine-grained computations on multiple processors.

6.2 Speculative Co-processing

6.2.1 Possibly Parallel Regions

The PPR markers are written as *BeginPPR(p)* and *EndPPR(p)*, where p is a unique identifier. While multiple *BeginPPR(p)*may exist in the code, *EndPPR(p)*must be unique for the same identifier. At a beginning marker, co-processing forks a process that jumps to the matching end marker and starts speculative execution there. The matching markers can only be inserted into the same function. The exact code sequence in C language is as follows.

- *BeginPPR(p)*: if (BeginPPR(p)==1) goto EndPPR_p;
- *EndPPR(p)*: EndPPR(p); EndPPR_p:;

As PPR markers suggest possible behavior, there is no guarantee on the order of their execution. A beginning marker may not be followed by its matching end marker, or an end marker may occur before any beginning marker. Co-processing constructs a sequence of zero or more non-overlapping PPR instances at run time. At any point t, the next PPR instance starts from the first beginning marker operation *BeginPPR(p)* after t and ends at the first end marker operation *EndPPR(p)* after the *BeginPPR(p)*. For example, assume there exist two PPR regions in the code, p and q, and let their markers be p_b, p_e, q_b, and q_e. If the execution, from the start s, produces the following trace (marked with their order in the superscript)

$$s\ p_b^1\ q_e^2\ p_e^3\ p_e^4\ q_b^5\ p_b^6\ q_e^7$$

Co-processing identifies two PPR instances: one from the first beginning marker p_b^1 to its first matching end p_e^3 and the other from the next beginning marker q_b^5 to its matching end q_e^7. The remaining parts of the trace, from s to p_b^1 and from p_e^3 to q_b^5, are executed sequentially. The second PPR instance will be run speculatively, and for the result to be correct, the parallel execution and the would-be sequential execution must both reach *EndPPR(q)*, which is unique in the code.

PPR markers can be used to bracket a region of the loop body to indicate that the regions can be run in parallel (while the code outside region is sequential), as shown in Figure 6.1 (a). This is an instance of pipelined parallelism. Many loops have a sequential part either due to its function, for example, the reading of a task queue, or due to its implementation, for example, the increment of a loop counter. PPR allows a programmer or a profiler (Section 6.2.4) to mark the likely boundaries between the sequential and parallel regions of a loop. The system automatically communicates data and detects dependence violations at run-time. PPR markers can also be used to indicate that two or more regions of code are likely parallel, as in Figure 6.2 (b).

The scope of PPR regions is dynamic and flat. This is in contrast to most parallel constructs, which have static and hierarchical scopes. Co-processing uses dynamic scopes to support high-level, coarse-grain tasks with a complete correctness guarantee. Coarse-grain tasks often execute thousands of lines of code, communicate through dynamic data structures, and have dynamic dependences and non-local control flows. Co-processing tolerates incorrect marking of parallelism in unfamiliar code. The PPR markers can be inserted anywhere in a program and executed in any order at run-time.

Next, we describe how to ensure correctness automatically once the PPR regions are marked.

6.2.2 Correctness

Co-processing guarantees that the same result is produced as in the sequential execution. To guarantee correctness, it divides the runtime data of a program into three disjoint groups: shared, checked, and private. More formally, we say $Data = (D_{shared}, D_{checked}, D_{private})$, where $D_{shared}, D_{checked}$, and $D_{private}$ form a partition of all data. This section first describes these three types of data protection with a running example in Figure 6.3 and a summary in Table 6.1. Then is a proof of correctness as a global property of the localized checks. Last is a comparison with existing methods.

For the following discussion we consider two Unix processes—the *main process* that executes the *current PPR* instance, and the *speculation process* that executes the *next PPR* instance and the code in between. The cases for k ($k > 1$) speculation processes can be proved inductively by validating the correctness for the $k-1$ case and then treat the kth process as the speculation and the earlier computation as the main process.

6.2.2.1 Three Types of Data Protection

Page-based protection of shared data All heap data by default are shared at *BeginPPR* by default. Co-processing preserves all run-time dependences on shared data at a page-level granularity. An example is the variable $shared$ in Figure 6.3. It holds the root of a tree that each PPR instance may use and modify. Page-based protection allows concurrent executions as long as the nodes on the same page do not cause a dependence violation. Many programs initialize and grow some large dictionary data structures. The shared-data protection allows non-conflicting access by both PPRs.

```
shared = ReadTree();
...
while (...) {
 ...
 BeginPPR(1)
 ...
 if (...)
  private = Copy(checked++)
  Insert(shared, new Node(private))
 ...
 if (!error) Reset(checked)
 ...
 EndPPR(1)
 ...
}
```

Figure 6.3: Examples of shared, checked, and private data

Table 6.1: Three types of data protection

type	shared data D_{shared}	checked data $D_{checked}$	(likely) private data $D_{private}$
protection	Not written by *main* and read by *spec*	Value at *BeginPPR* is the same as at *EndPPR*in *main*. Concurrent read/write allowed	no read before 1st write in *spec*. Concurrent read/write allowed
granularity	page/element	element	element
needed support	compiler, profiler run-time	compiler, profiler run-time	compiler (run-time)
target data	global vars, heap	global vars	stack, global vars

By using Unix processes, co-processing eliminates all anti- and output dependences through the replication of the address space. It detects flow dependences at run-time and at page granularity using OS-based protection. At *BeginPPR*, the program places shared data on a set of memory pages, turns off write permission for the current PPR and read/write permission for the next PPR, and installs customized page-fault handlers that open the permission for read or write upon the first read or write access. At the same time, the handler records which page has what type of access by which process. The speculation fails if and only if a page is written by the current PPR and accessed by the next PPR. All other access patterns are permitted If speculation succeeds, the modified pages are merged into a single address space before the execution continues.

Page level protection leads to false alerts. We can alleviate the problem by using profiling analysis to identify the global variables and dynamic allocation sites that cause false alerts and then use compiler and run-time support to allocate them on different pages. In the current implementation, we allocate each global variable on its own page(s). The shared data is never mixed with checked and private data on the same page, although at run time newly allocated heap data are private at first and then converted to shared data at *EndPPR*, as we will explain later.

Selective value-based checking Access-based checking is sufficient, but not necessary, for correctness. Consider the variable $checked$ in Figure 6.3. If the first conditional is frequently true then both the current and next PPR will modify and read $checked$. These accesses will then lead to a flow dependence between the PPRs. On the other hand, if the error condition is typically false, then the value of $checked$ is reset and will be the same at each *BeginPPR*. We refer to this situation as a *silent dependence* because the value from preceding writes is killed by the reset operation, and the flow dependence has no consequence and can be ignored, as we prove formally shortly after.

Most silent dependences come from explicit reinitialization. For example, the Gcc compiler uses a variable to record the loop-level of the current code being compiled. The value returns to zero after compiling a function. We classify these variables as *checked data*, which tend to take the same value at *BeginPPR* and *EndPPR*, in other words, the current PPR execution has no visible effect on the variable, as far as the next PPR instance is concerned.

There is often no guarantee that the value of a variable is reset by *EndPPR*. For example in Figure 6.3, if there is an error then there may be a real flow dependence between the two PPR instances. In addition, $checked$ may be aliased and modified after the reset, and the reset may assign different values at different times. Hence runtime checking is necessary.

Co-processing allocates checked variables in a region, makes a copy of their value at the *BeginPPR* of *main*, and checks their value at the *EndPPR* of *main*. In the current implementation, checked data must be global variables, so their size is statically known. Checked data are found through profiling analysis (described more in Section 6.2.4), which identifies variables whose value is likely to be constant at the PPR boundaries. Even if a checked variable does not return to its initial value in every PPR instance, co-processing still benefits if the value remains constant for just two consecutive PPR instances only.

Private data The third group, private data, is those that are known not to cause a conflict. In Figure 6.3, if *private* is always initialized before it is used, the access in the current PPR cannot affect the result of the next PPR, so the dependence can be ignored.

Private data come from three sources. The first is the program stack, which includes local variables that are either read-only in the PPR or always initialized before use. Intra-procedure data flow analysis is adequate for most programs. When the two conditions cannot be guaranteed by compiler analysis, for example, due to unknown control flow or the address of a local variable escaping into the program heap, we redefine the local variable to be a global variable and classify it as shared data. This solution does not work if the local variable is inside a recursive function, in which case we simply disable co-processing. This restriction applies only to procedures that appear on the call chain up to *BeginPPR*. Taking the address of local variables for recursive functions called from a PPR is permitted.

The second source is global variables and arrays that are always initialized before the use in the PPR. The standard technique to detect this is interprocedural kill analysis [Allen and Kennedy, 2001]. In many programs, the initialization routines are often called immediately after *BeginPPR*. However, a compiler may not always ascertain all cases of initialization. Our solution is for the system to automatically place each data element onto a separate page and treats it as shared until the first access. For aggregated data, the compiler automatically inserts calls after the initialization assignment or loop to classify the data as private at run time. Any access by the speculation process before the initialization aborts the speculation. Additionally, we allow the user to specify the list of variables that are known to be written before read in PPR. These variables are protected until the first write. We call this group *likely private data*.

The third type of private data is newly allocated data in a PPR instance. Before *BeginPPR*, the control process reserves a region of memory for the speculation process. Speculation would abort if it allocates more than the capacity of the region. The main process does not allocate to the region, so at *EndPPR*, its newly allocated data can be merged with the data from the speculation process. For programs that use garbage collection, we control the allocation in the same way but delay the garbage collection until after *EndPPR*. If any GC happens in the middle, it will cause the speculation to fail because of the many changes it makes to the shared data.

6.2.2.1.1 Synopsis The three data protection schemes are summarized and compared in Table 6.1. We make a few observations. First, they share the following property: The meta-data of *main* and *spec* is collected in isolation. Full correctness is checked after both processes finish. The strong isolation means that *the correctness of the system does not depend on communication during the parallel execution.* The shared data – whose protection is access based – and checked data – whose protection is value based – have a significant overlap, which are the data that are either read only or untouched by *main* and *spec*. We classify them as checked if their size is small; otherwise, they are shared. A problem is when different parts of a structure or an array require different protection schemes. Structure splitting, when possible, may alleviate the problem. Section 6.2.3 describes how we hide most of the protection overhead.

Section 6.2.4 describes how we classify all program data into these three groups. Section 6.3.1 describes the compiler support for data placement.

6.2.2.2 Correctness of Speculative Co-processing

It is sufficient to prove the correctness of a single instance of parallel execution between *main* and *spec*. We first define an abstract model of an execution.

memory V_x: a set of variables. V_{all} is the complete set (memory).

memory state S_V^t: the content of V at time t. For ease of reading we use S_x^t (rather than $S_{V_x}^t$) for the sub-state of V_x at t.

instruction r_x: an instruction of the program. Here we consider the markers of two PPRs, p and q, whose markers are p_b, p_e, q_b, and q_e. The two can be the same region.

execution state (r_x, S_V^t): a point in execution where the current instruction is r_x and the memory is S_V^t.

execution $(r_1, S_{all}^{t1}) \overset{p}{\Longrightarrow} (r_2, S_{all}^{t2})$: a continuous execution by process p from instruction r_1 and memory state S_{all}^{t1} to the next occurrence of r_2 with an ending state of S_{all}^{t2}.

If a parallel execution passes the three data protection schemes described before, all program variables in our abstract model can be partitioned into the following categories:

- V_{wf}: variables whose first access by $spec$ is a write. wf stands for write first.
- V_{excl_main}: variables accessed only by $main$.
- V_{excl_spec}: variables accessed only by $spec$.

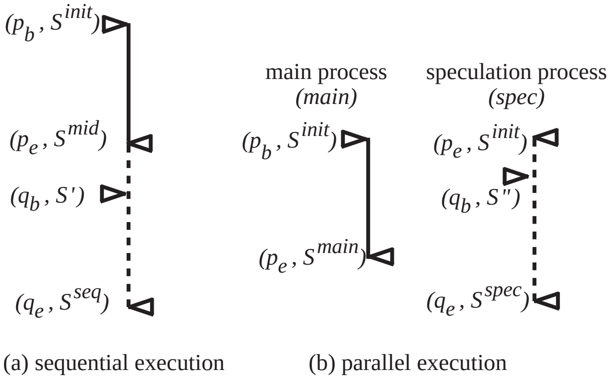


Figure 6.4: The states of the sequential and parallel execution. The symbols are defined in Section 6.2.2.2

- V_{chk}: the remaining variables. chk stands for checked.

$$V_{all} - V_{wf} - V_{excl\_main} - V_{excl\_spec}$$

Examining Table 6.1, we see that D_{shared} contains data that are either accessed by only one process (V_{excl_main} and V_{excl_spec}), written before read in *spec* (V_{wf}), read only in both processes (V_{chk}), or not accessed by either (V_{chk}). $D_{private}$ contains data either in V_{wf} or V_{chk}. $D_{checked}$ is a subset of V_{chk}. In addition, the following two conditions are met upon a successful check.

1. $main$ reaches the end PPR marker p_e after leaving the beginning marker p_b, and $spec$, after leaving p_e, encounters q_b and then q_e.
2. the state of V_{chk} is identical at the beginning and the end of $main$, that is, $S^{init}_{chk} = S^{main}_{chk}$.

To compare the result of the parallel execution with that of the sequential one, we examine their states, including the beginning, middle, and end of the sequential execution, S^{init} at p_b, S^{mid} at p_e, and S^{seq} at q_e; the start and end of *main*, S^{init} at p_b and S^{main} at p_e; and those of *spec*, S^{init} at p_e and S^{spec} at q_e. These states are illustrated in Figure 6.4.

Let the final state of the parallel execution be $S^{parallel}$ at q_e. $S^{parallel}$ is a combination of S^{main} and S^{spec} upon successful speculation. In particular,

$$S^{parallel} = S^{spec}_{all-excl\_main} + S^{main}_{excl\_main}$$

In words, the final state is the result of speculation plus the modified data in *main*.

We now quickly specify the instruction model and then move to the main proof. We define each operation r_t by its inputs and outputs. All inputs occur before anything is output. The inputs are a set of read variables $R(r_t)$. The outputs include a set of modified variables $W(r_t)$ and the next instruction to execute, r_{t+1} [1].

Theorem 1 (Correctness). *If* spec *reaches the end marker of the second PPR instance* q_e*, and the protection in Table 6.1 passes, the sequential execution would also reach* q_e*. Furthermore, the ending state of the sequential execution is identical to that of the parallel execution,* $S^{seq} = S^{parallel}$*, assuming that both start with the same state,* S^{init} *at* p_b*.*

Proof Consider the speculative execution, $(p_e, S^{init}) \overset{spec}{\Longrightarrow} (q_e, S^{spec})$, which speculates on the sequential execution, $(p_e, S^{mid}) \overset{seq}{\Longrightarrow} (q_e, S^{seq})$. Note that both start at the end PPR marker. We denote the correct sequential execution as $p_e, r_1, r_2, \cdots$ and the speculative execution as $p_e, r'_1, r'_2, \cdots$. We prove by contradiction that every operation r'_t in the speculative execution must be "identical" to r_t in the sequential execution in the sense that r_t and r'_t are the same instruction, they input from and output to the same variables with the same values, and they move next to the same instruction r_{t+1}.

Assume the two sequences are not identical and let r'_t be the *first* instruction that produces a different value than r_t, either by modifying a different variable, the same

---

[1] An operation is an instance of a program instruction. For the simplicity of the presentation, we overload the symbol r_x as both the static instruction and the dynamic instance. We call the former an *instruction* and the latter an *operation*. For example, we can have only one instruction r_x but any number of operations r_x.

variable with a different value, or moving to a different instruction. Since r_t and r'_t are the same instruction, the difference in output must be due to a difference in the input.

Suppose r_t and r'_t read a variable v but see different values v and v'. Let r_v and r'_v be the previous write operations that produce v and v'. r'_v can happen either in $spec$ before r'_t or in $main$ as the last write to v. We show neither of the two cases is possible. First, if r'_v happens in $spec$, then it must produce the same output as r_v as per our assumption that r'_t is the first to deviate. Second, r'_v is part of $main$ and produces a value not visible to $spec$. Consider how v can be accessed. Since (r'_v is the last write so) v is read before being modified in $spec$, it does not belong to V_{wf} or V_{excl_main}. Neither is it in V_{excl_spec} since it is modified in $main$. The only case left is for v to belong to V_{chk}. Since $V_{chk}^{main} = V_{chk}^{init}$, the last write in $main$ "restores" the value of v to the beginning state where $spec$ starts and consequently cannot cause r'_t in *spec* to see a different value as r_t does in the sequential run. Therefore r_t and r'_t cannot have different inputs and produce different outputs, and the speculative and sequential executions must be identical.

We now show that $S^{parallel}$ is correct or $S^{parallel} = S^{seq}$. Since $spec$ reads and writes correct values, V_{wf}, V_{excl_spec}, and the accessed part of V_{chk} are correct. V_{excl_main} is also correct because of the copying of the their values at the commit time. The remaining part of V_{chk} is not accessed by $main$ or $spec$ and still holds the same value as S^{init}. It follows that the two states $S^{parallel}$ and S^{seq} are identical. ▮

The style of the previous proof is patterned after the proof of the Fundamental Theorem of Dependence [Allen and Kennedy, 2001]. The conclusion rules out a common concern with value-based checking, where the value flow might produce a dependence before the last write. In co-processing, the three checking schemes work together to ensure no such case is possible.

6.2.2.3 Novel Features

Most existing speculation methods use what we term as weak isolation because the program data or the system meta data are concurrently accessed by parallel threads. In addition, the correctness checking happens while speculation continues. Weak isolation allows more dynamic parallelism at the risk of race conditions in the system and the user program. The problem is complicated by memory consistency problems due to the reordering of memory operations due to the compiler and the hardware and by the value-based checking in aggressive speculation support. Threads lack a well-defined memory model [Boehm, 2005]. Specific solutions are developed for the ABA problem in DSTM [Herlihy et al., 2003], for re-checking value prediction results in hardware [Martin et al., 2001], and for avoiding race conditions in software speculation (that does not use value-based checking) [Cintra and Llanos, 2005].

In co-processing, data are logically replicated, and the updates of data and meta-data by *main* and *spec* are completely separated from each other. The concluding correctness check is conclusive, as shown by Theorem 1. Consequently, no concurrency error may arise during the parallel execution and the correctness check. The compiler and hardware are free to reorder program operations as usual. While strong isolation does not support dynamic parallelism as efficiently as weak isolation, its simplicity suits co-processing, which uses extra processors to improve over the fastest sequential time on computations that may be parallelizable.

Most previous techniques monitor data at the granularity of array elements, objects, and cache blocks; co-processing uses pages for heap data and elements for global data. It uses Unix's forking mechanism and paging support. It allows monitoring of *all* global and heap data, reduces the monitoring cost to one or two page faults per page of data, and needs only a negligible amount of shadow data relative to the size of program data. The copy-on-write mechanism creates copies for modified data (for eliminating non-flow dependences and for possible rollbacks) on demand and in the background.

Paging has two downsides. The first is the cost of setting up permissions and handling page faults, though this will be amortized if PPR instances are large. The second is false sharing, which can be alleviated by data placement, as we discuss when describing the implementation.

Value-based checking is different from value-specific dynamic compilation (for example in DyC [Grant et al., 1999b]), which finds values that are constant for a region of the code rather than values that are the same at specific points of an execution (and can change arbitrarily between these points). It is different from a *silent write*, which writes the same value as the previous write to the variable, and from hardware-based value prediction, where individual values are checked for every load [Martin et al., 2001]. Our software checking happens once per PPR for a global set of data, and the correctness is independent of the memory consistency model of the hardware.

6.2.3 Performance

The parallel ensemble of processes hides most protection overhead off the *critical path*, which we define as the worst-performance execution where all speculation fails and the program runs sequentially.

6.2.3.1 Parallel Ensemble

We consider the case of using only one speculation process for co-processing. It needs four Unix processes. Each holds a logically separate copy of the program's address space and communicates through explicit communication. At any given time at most two processes are active. In general, k-process speculation requires $k+1$ processors (and $k+2$ processes).

The program starts as the *control* process. When reaching *BeginPPR*, *control* creates *main* and *spec*. The former executes the current PPR instance, while the latter jumps to the end marker and speculatively executes the next PPR instance (see Section 6.2.1

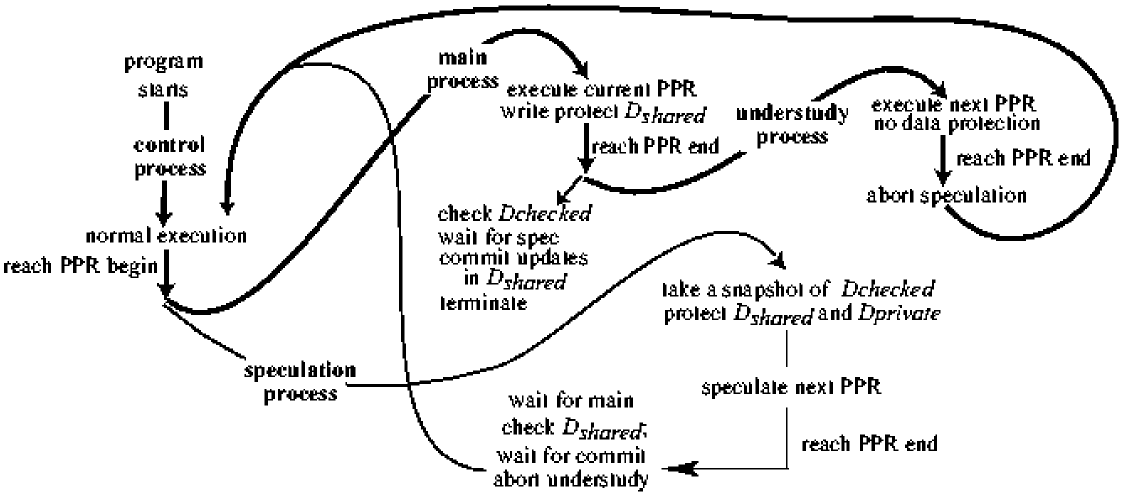


Figure 6.5: The parallel ensemble includes the control, main, speculation, and understudy processes. Not all cases are shown. See Table 6.2 for actions under other conditions.

for the definition of a PPR instance). When *main* reaches the end PPR marker, it immediately starts the *understudy* process, which re-executes the next PPR instance (not speculatively). Depending on whether speculation succeeds before *understudy* finishes, either *understudy* or *spec* becomes *control*, and is ready to start the next cycle of speculation. The diagram in Figure 6.5 shows the parallel ensemble. We first discuss the overhead in the first three processes and then turn the attention to the understudy process.

For shared data D_{shared}, *main* turns off write permission at *BeginPPR* while *spec* turns off read *and* write permission. They install a customized page-fault handler. The handler serves two purposes. First, the handler enables the page to be read from at the time of the first read, and to be written at the first write. Second, the handler records in an access map which page has what type of access by which process. When both processes finish, the two access maps are compared to check for a flow dependence. Upon commit, *main* copies all modified pages to *spec*.

Checked data $D_{checked}$ are protected in three steps. First, *spec* take a snapshot of $D_{checked}$. Second, *main* takes another snapshot when reaching *EndPPR* and compares it with the first one. In our implementation, the size of checked data is bounded and determined through profiling analysis, and the compiler groups the checked data into a contiguous memory region for fast copying and comparison at run time. Finally, likely private data are read and write protected in *spec* when it starts. The permission is opened either by a page fault or by a run-time call after the data is initialized.

Data copying takes time and may hurt locality. However, the locality in the same PPR instance is preserved. The footprint of co-processing is larger than the sequential run because of the replication of modified data. However, the read-only data is not copied and consequently will be shared by all four processes in main memory and in shared level two or three cache (that is physically indexed). As a result, the footprint is likely much smaller than running two copies of the program.

6.2.3.2 Understudy Process

As discussed in Section 6.2.2.3, co-processing cannot certify speculation results until both *main* and *spec* finish because of their strong isolation. For shared data, for example, a conflict may occur with the last write in *main* or the last read in *spec*. For large and complex programs, incorrect speculations may behave in an unpredictable manner: it may follow an incorrect path, execute a different PPR instance, exit, loop infinitely, or cause a segmentation fault.

Instead of waiting for *spec*, *main* starts *understudy* immediately upon finishing the current PPR instance and begins a two-way race between *understudy* and *spec*. As shown in Figure 6.5, if *spec* reaches *EndPPR* and finishes checking and committing changes before the understudy reaches *EndPPR*, *spec* kills the understudy and becomes the next *control*. However, if speculation misbehaves or takes too long to commit, the understudy will reach *EndPPR* first, abort *spec*, and continue as the next *control*.

The two-way race is a team race. Team understudy, which also includes *control* and *main*, represents the worst-case performance or the critical path. If all speculation fails, the three processes sequentially execute the program. The overhead on the critical path includes only the forking of *main* and *understudy* and the page-based write monitoring by *main*. The run-time cost is one page fault per page of shared data modified by *main*. There is no overhead associated with *understudy*. *All other overheads, forking and monitoring* spec, *taking and comparing snapshots, checking and committing speculation results, are off the critical path.* As a result, when the granularity of the PPR is large enough, the worst-case execution time of co-processing should be almost identical to that of the unmodified sequential execution. On the other hand, whenever a speculation process succeeds, it means a faster finish than the understudy and therefore a performance improvement over the sequential execution.

The performance benefit of *understudy* comes at the cost of potentially redundant computation. However, the incurred cost is at most running each speculatively executed PPR instance for the second time, regardless of how many PPR instances are speculated at a time.

With *understudy*, the worst-case parallel running time is equal to the best-case sequential time. One may argue that this can be easily done by running the sequential version side by side in a sequential-parallel race. The difference is that co-processing is a *relay race* for every two PPR instances. At the whole-program level it is sequential-parallel collaboration rather than competition because the winner of each relay joins together to make the co-processing time. Each time counts when speculation runs faster, and no penalty when it runs slower. In addition, co-processing allows read-only data shared in cache and memory, while multiple sequential runs do not. Finally, running two instances of a program is not always possible for a utility program, since the communication with the outside world often cannot be undone. In co-processing, unrecoverable I/O and system calls can and should be placed outside the parallel region.

Table 6.2: Co-processing actions for unexpected behavior

behavior	prog. exit or error	unexpected PPR markers
control	exit	continue
main	exit	continue
spec	abort speculation	continue
understudy	exit	continue

6.2.3.3 Expecting the Unexpected

The control flow in Figure 6.5 shows the expected behavior when an execution of PPR runs from *BeginPPR* to *EndPPR*. In general, the execution may reach an exit (normal or abnormal) or an unexpected PPR marker. If the current PPR instance is started with *BeginPPR(p)*, the expected marker is *EndPPR(p)*. Other markers such as *BeginPPR(p)* and markers of a different PPR, are unexpected. Unexpected behavior does not mean incorrect behavior. A program may execute PPR markers in any order. Table 6.2 shows the actions for *control*, *main*, *spec*, and *understudy* when encountering an exit, error, or unexpected PPR markers.

The abort by *spec* in Table 6.2 is conservative. For example, speculation may correctly hit a normal exit, so an alternative scheme may delay the abort and salvage the work if it turns out correct. We favor the conservative design for performance. Although it may recompute useful work, the checking and commit cost cannot delay the critical path.

The speculation process may also allocate an excessive amount of memory and attempt permanent changes through I/O and other OS or user interactions. The latter cases are solved by aborting the speculation upon file reads, system calls, and memory allocation over a threshold. The file output is buffered and is either written out or discarded at the commit point. Additional engineering can support regular file I/O. The current implementation supports stdout and stderr for debugging (and other) purposes.

6.2.4 Programming with PPR

We use offline profiling to find the possible parallel regions (PPRs). It identifies the high-level phase structure of a program [Shen et al., 2004a] and uses dependence profiling to find the phase with the largest portion of run-time instructions that can be executed in parallel as the PPR. At the same time, program data are classified into shared, checked and private categories based on their behavior in the profiling run. For lack of space, we will leave the detailed description to a later technical report.

Co-processing can also be added by a programmer. The programming interface has three parts. The first is the computation interface by which a programmer specifies PPRs using the two markers. The second is the data interface for the programmer to help the system classify all data as shared, checked, or private. Static variables are classified by the compiler analysis. Global and heap variables are considered shared by default. The data interface allows a user to specify a list of global and static variables that are write first (privatizable) in each PPR instance. The data interface supports the specification of checked data indirectly because a programmer can identify the value of the checked variable and insert assignment explicitly at the PPR boundary.

The write-first list opens the possibility of incorrect parallel execution when a variable is incorrectly classified as write first. For scalar variables the system can treat them as likely private data and check for unexpected access at run time. For aggregated data, the system cannot easily check. The programmer should test the system by re-initializing write-first variables at *BeginPPR* (possibly with random values to increase the chance of catching an error) and executing the program sequentially. If the output is not expected, the programmer can find the problem by debugging the sequential code. In general, the programmer should ensure that the write-first list is correct for all inputs. For any specific input, if the sequential program runs correctly, co-processing is guaranteed to return the same result.

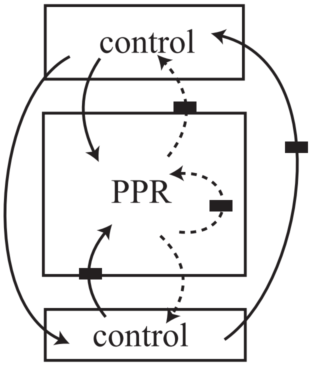
The third component of the interface is the run-time feedback to the user. When speculation fails, the system outputs the cause of the failure, in particular, the memory page that receives conflicting accesses. In our current implementation, global variables are placed on separate memory pages by the compiler. As a result, the system can output the exact name of the global variable when it causes a conflict. A user can then examine the code and remove the conflict by making the variable privatizable or moving the dependence out of the parallel region.

Three features of the API are especially useful for working with large, unfamiliar code. First, the user does not write a parallel program and never needs parallel debugging. Second, the user parallelizes a program step by step as hidden dependences are discovered and removed one by one. Finally, the user can parallelize a program for a subset of inputs rather than all inputs. The program can run in parallel even if it has an unknown number of latent dependences.

6.2.4.1 Profiling Support

A *possible parallel region (PPR)* is the largest region of a program that is likely parallel. Obviously the region should be selected carefully to avoid including irrevocable operations. A more difficult challenge is to select a likely parallel region. In the processing loop of a large utility program, every statement that executes once and only once in an iteration is a possible place for inserting region markers. The purpose of parallelism analysis is to consider all candidates and select the best region. For this paper we do not consider multiple PPRs in the same loop or nested (recursive) PPRs, although we believe that both can be used in co-processing with additional finesse.

We assume that the processing loop is known. One semi-automatic technique is active training, which first uses a sequence of identical requests to expose high-level phase structure and then uses real inputs to capture common sub-phases [Shen et al., 2004a]. Active profiling does not require a user to know anything about the program code. Alternatively, a user can manually identify the main processing loop and then



An illustration of PPR analysis

Loop-carried dependences are marked with a bar. Dotted dependences, if present, cause speculation failure.

Figure 6.6: Profiling analysis for finding the PPR

invoke the analysis to find common sub-steps. The result is a set of phase markers that are always executed in the same order in every processing step in all training runs. Note that the marker locations may spread throughout a program. Except in the simplest programs, the marker locations do not reside in the same function and are not executed in the order of their appearance in the source code.

Given n marker locations, each loop is broken into n primitive regions. Dependence profiling records the dependences between any two primitive regions and aggregates the results from all training runs. If we view each primitive region as a single statement, we can borrow the nomenclature of loop dependence [Allen and Kennedy, 2001]. The dependence between any two regions can be either loop independent or loop carried. Loop carried dependences may have a constant or a variable distance. We consider only flow dependences.

There are $\binom{n}{2} = \frac{n(n-1)}{2}$ candidate PPR regions, each sections the processing loop into three parts with two regions (we call control) on either side. Figure 6.6 shows an execution view, where each section is continuous and appears in order. While twelve types of cross-section dependence may happen in a three-section loop, the figure shows only common types for simplicity. Three types that would prohibit parallelization are shown with dotted edges, while loop-carried dependences are marked with a bar. The parallelism of a candidate region is then measured by the frequency of instances where the three dotted types are absent and by the number of run-time instructions in these instances. The best candidate is the one that gives the largest portion of run-time instructions that can be executed in parallel.

Currently we use brute force and test all regions. Alternatively we can use a graph model, where each primitive region is a node. The worst case complexity is the same since the number of edges may be $O(n^2)$.

In the example in Figure 6.3, a marker may be inserted before every top-level statement in the while loop. Consider the assignment of Z at the beginning of the loop. Assuming no other assignment of Z, then BeginPPR would be placed either before or after the assignment. The analysis picks the earlier spot because the size of PPR is larger. Once PPR is determined, the analysis classifies each global variable as shared, checked, or private data.

The classification depends on the choice of PPR. In the example, Z is private when BeginPPR proceeds the assignment but it would be checked if BeginPPR followed the assignment. While profiling analysis is sufficient to determine the first two groups, compiler analysis, as described in Section 6.2.2, is needed for the third group, for example, ensuring Z is always initialized before used. Besides those, the profiling analysis finds the variables that are never read before being written in all PPR instances through the profiling run. The run-time system doesn't need to monitor those variables except ensuring their write-first property by closing their read and write permissions at EndPPR of the speculative process and opening the permission at the first write operation. Any unpermitted operation causes the speculation process to abort.

Profiling analysis has long been used to measure the behavior of complex programs. Many studies have examined the potential of parallelism and locality. Coarse-grain parallelism analysis has three unique features. The first is large granularity. A PPR may span many loops and functions in thousands of lines of code. Much code may be used in both the sequential and parallel parts. The second is cross-input behavior. The set of markers are valid across all training runs, so the parallelism results are also correlated across all runs. Last is the integration with behavior protection. The usage pattern is used to classify data into different protection groups. As a result, the system exploits parallelism when the behavior is expected but reverts to sequential execution with little additional overhead when the behavior is unexpected.

In addition to profiling and compiler analysis, co-processing uses the run-time bookkeeping to report the cause whenever a speculation fails. In particular, it places all global variables on a separate page, so it can pinpoint the exact variable. As we will discuss later, this feature is especially useful for the manual adaptation of a program for co-processing. The run-time feedback leads a user directly to the few key variables and routines amidst a sea of other code and data not relevant to coarse-grain parallelism.

Co-processing depends on the granularity of PPRs. In addition to off-line profiling, on-line analysis can be used to disable speculation if a program does not show large, parallel regions or if data protection requires excessive overhead (for the speculation process). The run-time system monitors the granularity of the last PPRs possibly using hardware counters and the success rate of speculation, which can then be easily disabled by changing a flag. The remaining program is then executed by the control process without interruption.

6.3 Evaluation

6.3.1 Implementation

We have implemented the compiler support in Gcc 4.0.1, in particular, in the intermediate language, GIMPLE (based on static-single assignment [Cytron et al., 1991] similar to SIMPLE form [Hendren et al., 1992]), so the transformation is applied after high-level program optimization passes but before machine code generation. The main transformation is converting global variables to use dynamic allocation, so the run-time support can place them for appropriate protection. The compiler allocates a pointer for each global (and file and function static) variable, inserts an initialization function in each file that allocates heap memory for variables (and assigns initial values) defined in the file, and redirects all accesses through the global pointer. All initialization functions are called at the beginning of the main function. As we will see later, the indirection causes only marginal slowdown because most global-variable accesses have been removed or converted to (virtual) register access after the GIMPLE passes. Source-level indirection would be much more costly.

For parallelism analysis we also implemented an instrumentor, which collects complete data and instruction access traces for use by the behavior analyzer. It provides unique identifiers for instructions, data accesses, and memory and register variables, so the behavior analyzer can trace all data dependences and identify possible phase markers. We have implemented similar systems using two binary instrumentors, which do not require program source but offer no easy way of relocating global data, tracking register dependences, or finding the cause of conflicts at the source level.

For data protection, we have not implemented the compiler analysis for local variables. Instead the system privatizes all stack data. Global and heap variables are protected. Each global variable is allocated on a separate page(s), which reduces false sharing at a bounded space cost.

The run-time system is implemented as a statically linked library. Shared memory is used for storing snapshots, access maps, and for copying data at a commit. Five types of signals are used for process coordination, which we do not elaborate for lack of space, except for four points. First, there is no busy waiting from locks or semaphores. Second, only two signals, SIGSEGV and SIGUSR1 (for process ready), may happen on the critical path. In addition, the design guarantees forward progress, which means no deadlocks or starvation provided that the OS does not permanently stall any process. Finally, it was tricky to design and debug the concurrent system with four types of processes and 15 customized signal handlers. To improve efficiency, the implementation uses shared meta-data to pass information.

Experimental Setup

In our initial design, the program process and any other process did not stay alive longer than two PPRs (for one moment we thought we had an incredible speedup). Currently each program starts with a timing process, which immediately forks the first control process and waits until a valid exit is reached. We cannot collect user and system time for all processes, so we use wall-clock time of the timing process, which includes OS overheads in process scheduling. We use the smallest over three runs on an unloaded system.

For each program we measure four variations. The time of the unmodified original program is labeled *original*. The time of the sequential version before PPR insertion is labeled *sequential*. It differs from the original program in that all global variables have been changed to dynamic allocation (and separately placed on different memory pages at run time). For a program with manually inserted PPRs, the original version may be transformed by unrolling the processing loop (to increase the granularity). The sequential version includes the effect of all manual changes except for the PPR markers. The third is the worst-case time of co-processing, labeled *spec fail*. We artificially induce conflicts so speculation always fails. It gives the time of the critical path. Last is the *co-processing* time, which measures the improvement from the dynamic speculation. We use GNU Gcc 4.0.1 with “-O3” flag for all programs.

The test machine has two Intel Xeon 2 GHz processors with 512K L1 cache, 2GB Memory, and hyperthreading. The relative effect of co-processing is similar with and without hyperthreading. We report the running time with hyperthreading turned on, which sometimes makes a program marginally faster. We also tested a dual-processor Intel 1.27 GHz Pentium III workstation. Since co-processing mainly reduces CPU time, the effect is more dramatic on the slower machine (up to 40% slower than the 2GHz Xeon). We do not report Pentium III results for lack of space. However, we note that co-processing would be more effective on multi-core chips if the cores are made from simpler and slower CPUs.

6.3.2 Micro-benchmarks

We wrote two small programs and manually inserted PPR markers to examine the cost and benefit of co-processing over controlled parameters including the size of data, intensity of computation, and frequency of run-time conflicts. The next section shows the result on real programs, including the automatic insertion of PPRs.

6.3.2.1 Reduction

The reduction program initializes an array of n integers, performs k square-root operations, and adds the results together. The parallel version adds the numbers in blocks, each block is a PPR. To get the *spec fail* version, all PPRs use the same sum variable. The speculation always fails because of the conflict. In the *co-processing* version, we use a partial sum for each block and add them at the end.

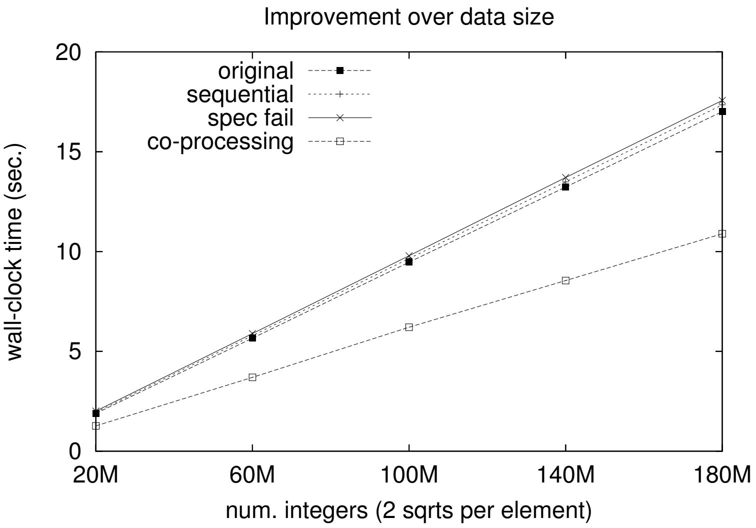


Figure 6.7: Co-processing performance for the reduction program

Figure 6.7 shows the performance when n increases from 20 million to 180 million in 40 million increments. The computation intensity, k, is two square-roots per element, and the speculation happens once (two blocks). In all versions, the time scales in a straight line with the data size. *Sequential* is 2% to 3% slower than *original* possibly due to the indirect access to the reduction variable. *Spec fail* adds another 1.5% to 3%. Since the overhead scales with the data size, most of it is the cost of the page fault incurred by the main process for each page of shared data. *Co-processing* improves the speed by 48%, 53%, 52%, and 55% for the four data sizes from the smallest to the largest (about one third reduction in running time).

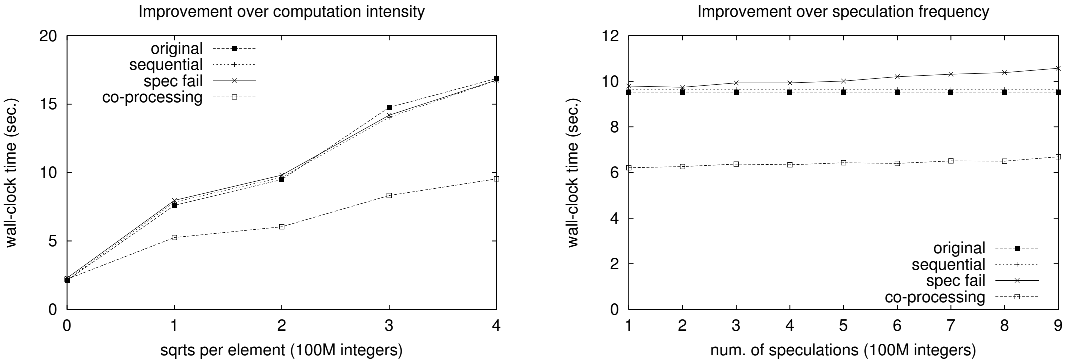


Figure 6.8: Co-processing performance for the reduction program

Next we vary the computation intensity and the frequency of speculation. The left half of Figure 6.8 shows the computation intensity k from zero to four square-roots per element. When there is no computation, the speed is bounded by the effective memory bandwidth, and co-processing is pure overhead. Compared to *original, sequential* is 3% slower, *spec fail* 7%, and *co-processing* 3.6%. When there is much computation ($k = 4$), *sequential* is 5% faster, *spec fail* 4%, and *co-processing* 77%. The right half of Figure 6.8 shows the effect of speculation granularity. For 100 million numbers and two square-roots per number, we reduce the block size to increase the number of blocks from 2 to 18 and hence the speculation attempts from 1 to 9 for the same amount of computation. *Original* and *sequential* have the same time in the figure because they do not speculate. The slowdown from *spec fail* (worst-case time) over *sequential* increases from 1.4% to 8.7%, and the improvement from co-processing decreases from 55% to 44%. For this workload on average, each additional speculation adds 1% time overhead and loses 1% performance gain.

6.3.2.2 Graph Reachability

Our second benchmark is a computation of graph reachability. For an undirected graph, a typical reachability test performs depth-first search on each node, marks reachable nodes, and skips the next one if it is marked. The amount of parallelism available is entirely input dependent: the test loop is fully parallel if the graph has no edge, but it is completely sequential if the entire graph is connected. For this test we use random graphs with different average node degrees. We create a favorable environment for co-processing—each node is 4KB in size, and a large amount of computation is performed for each connected component. Figure 6.9 shows the results when a random graph of 100 nodes has between 1 and 100 connected components. The solid curve shows the portion of nodes in the connected components found by speculation. The dotted curve shows the reduction in running time. The result shows that co-processing can exploit highly dynamic and input-dependent parallelism with no explicit parallel programming—no locks, semaphores, or barriers. A user writes a sequential program and then inserts PPR. For co-processing to be profitable however, we need large granularity, which may exist in large programs.

6.3.3 Application Benchmarks

6.3.3.1 Gzip v1.2.4 by J. Gailly

As a compression tool, *Gzip* takes one or more files as input and compresses them one by one using the Lempel-Ziv coding algorithm (LZ77). The version we use is 1.2.4 and comes from the SPEC 2000 benchmark suite. We did not specify “spec” so the program behaves as a normal compressor rather than a benchmark program (which

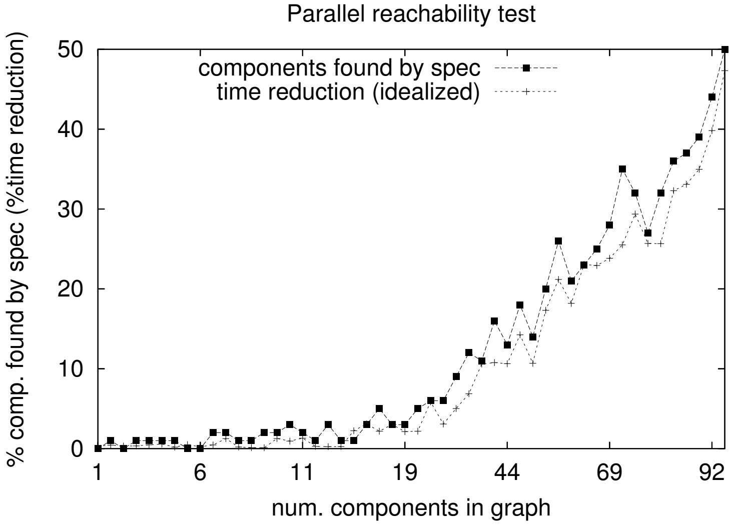


Figure 6.9: Co-processing performance for the reachability test

artificially lengthens the input by replication). The program has 8616 lines of C code. *BeginPPR* and *EndPPR* are automatically inserted before reading a file and after the output of the compressed file (for this one we allow file I/O in the PPR). As shown in Table 6.3, the analyzer identifies 33 variables and allocation sites as shared data, 78 checked variables (many are not used during compression), 33 likely private variables. Behavior analysis, in fact, detected flow dependences between compressions because the original *Gzip* reinitialized only part of the data structure before compressing another file. The values were used but seemed to have no effect. We changed the code to reinitialize these variables to 0. Compression returns identical results in all test inputs.

For each file, *Gzip* compresses one input buffer at a time and stores the results until the output buffer is full. We manually placed PPR around the buffer loop and specified the set of likely private variables through the program interface described in Section 6.2.4. The program returned correct results but speculation failed because of conflicts caused by two variables, “unsigned short bi_buf” and “int bi_valid”, as reported by the run-time feedback. The two variables are used in only three functions in a 205-line file. Inspecting code, we realized that the compression produced bits, not bytes, and the two variables stored the partial byte between compressing consecutive

Table 6.3: The size of different protection groups in the training run

Data groups		Gzip	Parser
shared data	num. objs.	33	35
	size(bytes)	210K	70K
	accesses	116M	343M
checked data	num. objs.	78	117
	size(bytes)	2003	5312
	accesses	46M	336M
(likely) private data	num. objs.	33	16
	size(bytes)	119K	6024
	accesses	51M	39M

buffers. The dependence was hidden below layers of code and among 104 global variables, but the run-time analyzer enabled us to quickly pin down the cause. We modified the code to compress buffers in parallel and concatenate the compressed bits afterwards.

Figure 6.10 shows the results of three sets of tests. The first is compressing 10 identical archive files, each a mix of text, Powerpoint and binary files. This is the best case for co-processing, and the compression runs 78% faster. The second is the set of five files in Spec2K ref input. Two files are compressed in parallel, which leads to a lower 16% improvement due to the different length of PPR instances. With an even length (when we replicate the five files), the improvement becomes 51%. The third input is an 84MB Gcc tar file. The intra-file co-processing speculates on 30MB of compression and improves the compression time by 34%.

Inter-file co-processing uses around 130KB additional memory in all executions, mostly for likely private data shown in Table 6.3. Intra-file co-processing uses 7.45MB (1865 replicated pages) additional memory, mostly for *spec* to buffer the compressed data for the input used. In addition, the program has 104 global variables, so the space overhead for page allocation is at most 104 pages or a half mega-byte for the sequential execution. The space cost of their run-time replication is already counted in the numbers above (130KB and 7.45MB).

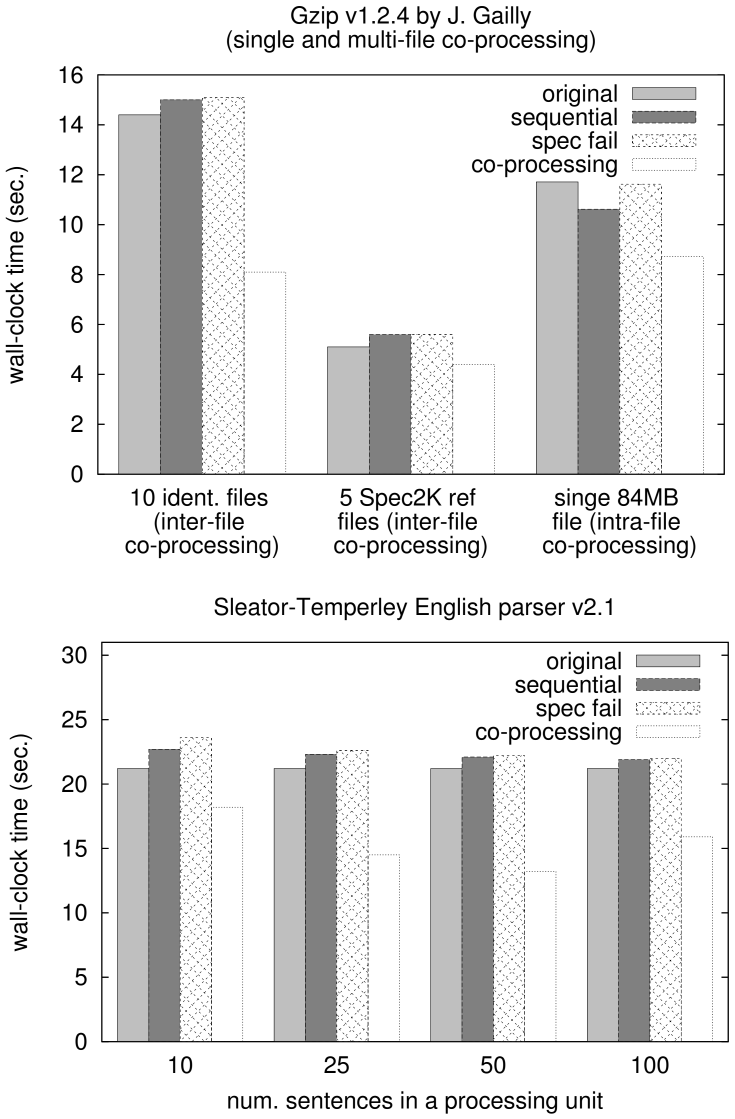


Figure 6.10: The effect of co-processing on *Gzip* and *Parser*

6.3.3.2 Sleator-Temperley Link Parser v2.1

According to the Spec2K web site, "The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. ... It is able to handle unknown vocabulary, and make intelligent guesses from context about the syntactic categories of unknown words." It is not clear in the documentation or the 11,391 lines of its C code whether the parsing of sentences can be done in parallel. In fact, they are not. One dependence we found during training comes from commands (mixed with sentences) that, for example, turn on or off the echo mode for printing parsed sentences.

The parallelism analyzer identifies the sentence-parsing loop. We manually strip-mine the loop to create a larger PPR. The data are then classified as in Table 6.3 automatically. During the training run, 16 variables are always written first by the speculation process during training, 117 variables always have the same value at the two ends of a PPR instance, and 35 variables are shared.

The lower graph of Figure 6.10 shows the performance on an input with 600 sentences. We tested different strip-mine sizes from 10 sentences to 100 sentences in each group. The group size has mixed effects on program performance. For *sequential* and *spec fail*, the largest group size gives the lowest overhead, 3.1% and 3.6% respectively. Co-processing improves performance by 16%, 46%, 61%, and 33% for the four group sizes. The best performance happens with the medium group size. When the group size is small, the relative overhead is high; when the group size is large, there are fewer PPR instances and hence more likely uneven-size PPRs. Finally, the space overhead of co-processing is 123KB, 100KB of which is checked data. The space overhead does not seem to change with the group size.

6.3.3.3 ATLAS by R. C. Whaley

The Automatically Tuned Linear Algebra Software (ATLAS) is one of the fastest library implementations of linear algebra routines [Whaley et al., 2001]. It is used widely by scientific programmers and included in larger systems such as Maple, MATLAB, and Mathematica. Using parameterized adaptation and source code adaptation, it generates different source code depending on the types of parameters of the input and the machine environment such as the data type, matrix size, cache size and length of floating point pipelines. The version of ATLAS built for our Pentium 4 machine contains 2,905 source files with 1,991,508 lines of code and 4,904 object files. Compilation was done with the following parameters: Posix thread support enabled, express setup enabled, maximum cache size of 4096 KB, and Level 1 BLAS tuning enabled. The total time for compilation was 2 hours and 42 minutes. The developer version, release 3.7.11, of the code is used which gives access to the most updated sources, which run faster in our experiment when compared to the stable version, release 3.6.0. On the test machine, base ATLAS is an order of magnitude faster than the naive implementation, whose speed is about 400 MFLOPS. The multi-threaded ATLAS comes with the 3.7.11 distribution. It is implemented by hand and tuned to use up to four threads on the test machine.

In this experiment we compare two parallel implementations of square matrix multiply: co-processed sequential ATLAS and threaded ATLAS. In co-processing, the user data is protected from conflicting accesses by the program and the library. The data inside the library needs no protection since each PPR process uses it as a sequential program does. We compare five versions of square matrix multiply: *base atlas* initializes two matrices and makes one call to sequential ATLAS, *threaded atlas* calls the parallel version, *base atlas* + *co-processing* computes the result matrix in three steps: the upper matrix and the lower matrix in two parallel regions and the middle section at the end (to avoid false sharing), *threaded atlas* + *co-processing* calls threaded ATLAS inside the parallel regions, and finally *co-processing spec fail* inserts an artificial conflict to cause co-processing to fail. For matrices of N^2 size, the performance is measured by $2 * N^3$ divided by the total wall-clock running time. The results are shown in Figure 6.11.

Co-processing runs slower than *base atlas* for N less than 800 where the speculation overheads outweigh the benefit of parallelism. After N reaches 2800 (and the time reaches 8.7 seconds), co-processing outperforms manually parallelized and tuned ATLAS by 2% and the base ATLAS by as much as 86%. The combined co-processing and threaded ATLAS runs correctly but has a lower parallel performance. The version *spec fail* performs significant slower than *base atlas* when N is below 1400, showing

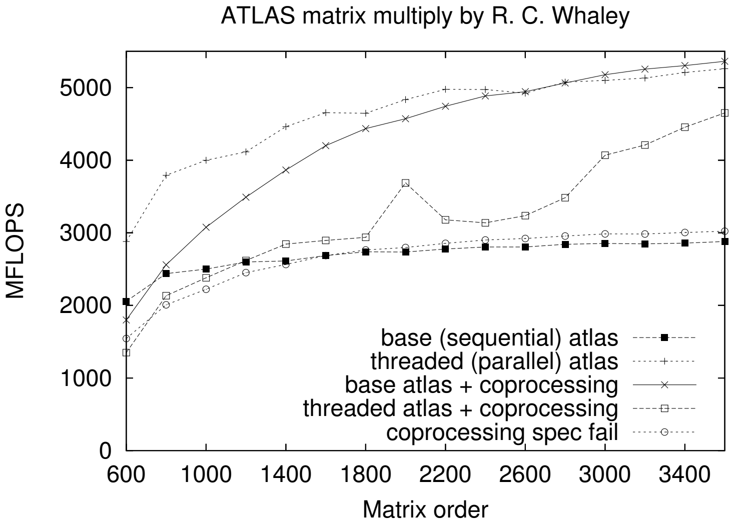


Figure 6.11: Co-processing performance with ATLAS

the effect of the speculation overhead when the running time is under 2 seconds. The two versions then run neck and neck for N up to 2000, after which *spec fail* wins a nose ahead. It seems that breaking the multiply into three pieces leads to faster sequential performance. The space overhead changes with the input. For the largest run, $N = 3600^2$, co-processing uses 45.2MB (11312 copied pages) additional memory, for mostly the half matrix being speculatively computed on.

An important observation is that the threaded ATLAS has less smooth performance than the base ATLAS, indicating that parallel tuning is more difficult for a programmer than sequential tuning. In contrast, the parallel performance of co-processing scales as smoothly as the sequential ATLAS, showing the advantage of co-processing based on the fastest (and often the best tuned) sequential code.

6.4 Related Work

A complete system like ours is undoubtedly built on the ideas of many earlier projects. For lack of space, we cite mostly software solutions but similar ideas of speculation and data monitoring have been explored extensively in hardware-based solutions.

Parallel languages The separation of performance and correctness is a key idea embodied in High Performance Fortran, where a user can specify data distribution but let the compiler parallelize the program [Forum, 1997; Allen and Kennedy, 2001]. In co-processing, a user specifies likely rather than definite parallelism. Pipelined parallelism in loops can be specified by the *doacross* construct, where specific data are posted by the producer iteration and waited by the consumer iteration [Cytron, 1986; Allen and Kennedy, 2001]. Most parallel languages have constructs for specifying parallel statements. Well-known examples include the parallel regions in OpenMP [OpenMP], transactions in transactional memory [Herlihy and Moss, 1993], and future in Multilisp [Halstead, 1985]. The future construct specifies that the result of a computation is not immediately needed until a later point, which can be inferred in pure functional languages as in Multilisp [Halstead, 1985] or explicitly marked by a programmer as the end of a transaction in transactional memory [Herlihy and Moss, 1993] or the *sync* point in Cilk [Frigo et al., 1998].

Like HPF and transactions, possibly parallel regions do not guarantee parallelism. PPR goes one step further because data sharing and synchronization points are implicit. The scope of a region is dynamic rather than static, so it allows arbitrary control flow in, out, and between parallel regions. On the other hand, parallel languages are often more expressive and can specify nested parallelism([Frigo et al., 1998; Moss, 2006] for examples) and exact data sharing, such as reduction and data copying in OpenMP [OpenMP] and typed and programmable specification in Jade [Rinard and Lam, 1998].

Dynamic and speculative parallelization The concept of data dependence was developed for parallelization (vectorization) by Lamport in the Parallelizer system, by Kuck and his colleagues in Paraphrase, and by Kennedy and others in Parallel Fortran Converter (PFC) [Allen and Kennedy, 2001]. Control dependence was developed by Ferrante et al [Cytron et al., 1991; Allen and Kennedy, 2001]. Static dependence checking can be overly conservative when two statements are mostly but not always independent and when the independence is too difficult to prove, especially between large code regions. Early dynamic checking techniques developed for array-based scientific programs include the inspector-executor for dynamic parallelization [Saltz et al., 1991] and the privatizing *doall* (PD) test for speculative parallelization [Rauchwerger and Padua, 1995]. The PD test has two separate phases: the marking of data access and checking for dependence. Later techniques speculatively privatize shared arrays (to allow for non-flow dependences) and combine the marking and checking phases [Gupta and Nim, 1998; Dang et al., 2002; Cintra and Llanos, 2005]. The technique of array renaming is generalized in Array SSA [Knobe and Sarkar, 1998]. Inspection is used to parallelize Java programs at run-time [Chan and Abdelrahman, 2004].

These techniques are more scalable than co-processing currently is. They address issues of parallel reduction [Gupta and Nim, 1998; Rauchwerger and Padua, 1995; Saltz et al., 1991] and different strategies of loop scheduling [Cintra and Llanos, 2005]. In co-processing, a user can enable parallel reduction by explicit coding.

Hardware-based thread-level speculation is among the first to automatically exploit loop- and method-level parallelism in integer code. In most techniques, the states of speculative threads are buffered and checked by monitoring the data writes in earlier threads either through special hardware additions to a processor [Sohi et al., 1995], bus snooping [Chen and Olukotun, 2003], or an extended cache coherence protocol [Steffan et al., 2005]. Since speculative states are buffered in hardware, the size of threads is limited to no more than thousands of instructions. A recent study classifies existing loop-level techniques as control, data, or value speculation and shows that the maximal possible speedup is 12% on average for SPEC2Kint even with no speculation overhead and unlimited computing resources [Kejariwal et al., 2006]. The limited potential suggests that the programmer support like ours is needed for speculative system to fully utilize multi-processor machines.

Speculative execution is closely related to methods of nonblocking concurrency control. Run-time dependence checking is an efficient (but not necessary) solution to ensure serializability, which is NP-hard in the general case [Papadimitriou, 1979]. An influential solution assigns a sequence number to each transaction and ensures the same result as the serialized execution [Kung and Robinson, 1981]. Transactional memory was originally proposed as a hardware mechanism to support nonblocking synchronization (by extending cache coherence protocols) [Herlihy and Moss, 1993]. It is rapidly gaining attention because of its potential to be a general and easy to use solution for concurrency [Grossman, 2006]. Various software implementations rely on transactional data structures and primitive atomic operations available on existing hardware [Harris and Fraser, 2003; Herlihy et al., 2003; Shavit and Touitou, 1997] (see [Marathe and Scott, 2004] for a survey). Many hardware-based TM systems have also been developed.

As discussed in more detail in Section 6.2.2.3, co-processing is different from most existing speculation techniques in three aspects: page-based monitoring, value-based checking, and strong isolation. Value-based checking allows co-processing in the presence of flow dependences, so it improves the basic dependence checking as used by existing software-based schemes [Chan and Abdelrahman, 2004; Cintra and Llanos, 2005; Gupta and Nim, 1998; Knobe and Sarkar, 1998; Rauchwerger and Padua, 1995; Saltz et al., 1991]. Strong isolation protects correctness but opens the possibility of speculation failure after the main process finishes. The understudy process is a novel solution to this problem. The understudy execution has no protection overhead except for forking and copying modified pages, which is necessary for its cancellation when speculation finishes early and correctly. Being able to rollback a safe execution to improve performance is an interesting feature of the system.

Run-time data monitoring For large programs using complex data, per-access monitoring causes slow-downs often in integer multiples, as reported for data breakpoints and on-the-fly data race detection, even after removing as many checks as possible by advanced compiler analysis [Mellor-Crummey, 1992; Perkovic and Keleher, 2000; Wahbe et al., 1993]. Dynamic co-processing cannot possibly afford such slowdown and be practical. Page-based data monitoring was used for supporting distributed shared memory [Li, 1986; Keleher et al., 1994] and then for many other purposes. Co-processing uses page-based monitoring for shared data to trade precision for efficiency (without compromising correctness). For likely private data and for checked data, it incurs only a constant cost per PPR. Most speculation overhead occurs on the speculative path. Only one page fault per modified page is incurred on the critical path. No other software systems we know has as low an amortized cost for loosely coupled parallelism.

6.5 Future Directions

Behavior-Oriented Parallelization can be extended in three directions: efficiency, scalability, and applicability. The current scheme hasn't exploited compiler analysis much. Potentially, a good dependence analysis should help reduce both profiling and run-time overhead.

For scalability, one possibility is to extend the co-processing to multi-processing, where more than one process could do speculative execution simultaneouly. An implementation difficulty is the increased potential race conditions among the processes. The second possibility is a distributed version of behavior-oriented parallelization, a scheme working on both SMP systems and clusters.

The current co-processing is designed and tested on utility programs with one PPR only. The extension to a wider range of applications with more PPRs remains a future research topic.

6.6 Summary

The paper has presented the design and evaluation of co-processing including the PPR markers for specifying likely parallelism, strong isolation for protecting shared, checked and likely private data, and the parallel ensemble for hiding the speculation overhead and ensuring that the worse parallel performance is as good as the best sequential performance. Our prototype includes a profiler, a modified GNU C Compiler, and a run-time system. On a dual-processor PC, co-processing shows expected properties when tested on micro-benchmarks and improves performance by 16% to 86% for two large integer applications and a scientific library.

Co-processing provides a new programming system. Known dependences, such as error handling and garbage collection, can stay in code as long as they happen rarely. Parallelization can be done in incremental steps by removing dependences one by one as detected by the run-time feedbacks. At no point does a programmer need parallel debugging.

Not all programs have loosely-coupled coarse-grain parallelism, although many utility programs do, at least for certain inputs. Process-based (behavior-level) parallelism complements finer-grained parallelism such as threads. In fact, a PPR may contain threaded code. Overall, these results show that co-processing offers cost-effective way to make use of coarse grain parallelism and to improve the best sequential implementation on existing parallel hardware with no explicit parallel programming.

7 Conclusions and Speculations

My advisor, Professor Chen Ding, once asked the students in a compiler class whether computer science is science. Students immediately asked for the definition of "science". The definition Professor Ding offered is similar to the following[1]:

> "Any system of knowledge that is concerned with the physical world and its phenomena and that entails unbiased observations and systematic experimentation. In general, a science involves a pursuit of knowledge covering general truths or the operations of fundamental laws."

Well, that helped, but not ultimately: the students' answers diverged from uniform uncertainty to polar opposites.

The arguments on the question are out of the scope of this thesis, but one of its ensuing questions is quite relevent: what is the relation between computer programs and the physical world? Within the last century, with no doubt, the former has quickly covered almost every aspect of the latter. Computer programs have composed a new world—the program world, which models and interacts with the physical world, and shows interesting analogies with the latter.

---

[1] From Encyclopedia Britannica: http://www.britannica.com

An N-body molecule system for example behaves differently in a different physical environment. Physics research tries to discover the underlying laws and predict the behavior of an arbitrary set of molecules in any environment. Program analysis, similarly, tries to discover the general behavior patterns of a program and predict its behavior for an arbitrary input in any running environment.

In the physical world, an object has both intra- and interactions. The intra-actions of the Earth form mountains; the interaction with the Sun brings us four seasons. A program has intra-actions among its data, functions and components; it also interacts with other programs, sometimes being constructive as providing or obtaining extra functionalities, sometimes being destructive as competing for limited resources.

Mother nature seeds the capability of learning, self-evolution and adaptation into every life, which is essential for the progress of the physical world. In my opinion, the program world needs such capability not less at all. A program, running in a different environment (machines, operating systems) with various inputs, mechanically follows the road map designed at its "birth" despite the actual environment, inputs and its running history: its one billion’th run on the same input is not any better than its first run. The program world needs intelligence.

Behavior-based program analysis is an exploration in that direction. This thesis describes three aspects: whole-program locality and affinity analysis, program phase analysis, and behavior-oriented parallelization. The first two start from cross-input prediction of the average behavior of a whole program, and extend to the prediction of large-scale dynamic behavior of program phases. Forecasting the future enables various adaptations like data reorganization and cache resizing. Behavior-oriented parallelization is a special kind of adaptation: a sequential program is (semi-)automatically given the ability to utilize multi-processors when necessary. These techniques reveal both spatial and temporal large-scale program patterns, which are not visible from individually analyzing program code, data, input, or running environment, but indispensable to creating an intelligent program world.

This research has established a new basis for intelligent programming systems, which introduce into a program the ability to automatically evolve its code and data and configure its running environment such that a better version of the program could dynamically match the input, behavior and system conditions.

Bibliography

Allen, F. and J. Cocke. 1976. A program data flow analysis procedure. *Communications of the ACM*, 19:137–147.

Allen, R. and K. Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers.

Almasi, G., C. Cascaval, and D. Padua. 2002. Calculating stack distances efficiently. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*. Berlin, Germany.

Anderson, J., S. Amarasinghe, and M. Lam. 1995. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Santa Barbara, CA.

Arnold, M. and B. G. Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah.

Bailey, D. 1992. Unfavorable strides in cache memory systems. Technical Report RNR-92-015, NASA Ames Research Center.

Balasubramonian, R., D. Albonesi, A. Buyuktos, and S. Dwarkadas. 2000a. Dynamic memory hierarchy performance and energy optimization. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.

Balasubramonian, R., D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. 2000b. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd International Symposium on Microarchitecture*. Monterey, California.

Balasubramonian, R., S. Dwarkadas, and D. H. Albonesi. 2003. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of International Symposium on Computer Architecture*. San Diego, CA.

Balasundaram, V., G. Fox, K. Kennedy, and U. Kremer. 1991. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Williamsburg, VA.

Batson, A. P. and A. W. Madison. 1976. Measurements of major locality phases in symbolic reference strings. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*. Cambridge, MA.

Beyls, K. and E.H. D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*.

Beyls, K. and E.H. D'Hollander. 2002. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*. Paderborn, Germany.

Boehm, Hans-Juergen. 2005. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference On Programming Language Design and Implementation*, pages 261–268.

Burd, T. and R. Brodersen. 1995. Energy efficient cmos microprocessor. In *The Proceedings of the 28th Hawaii International Conference on System Sciences*.

Burger, D. and T. Austin. 1997. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, Department of Computer Science, University of Wisconsin.

Burger, D., S. Keckler, M. Dahlin, L. John, C. Lin, K. McKinley, C. Moore, J. Burrill, R. McDonald, and W. Yoder. 2004. Scaling to the end of silicon with edge architectures. *IEEE Computer*, 37(7):44–55.

Burke, M. and R. Cytron. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction*. Palo Alto, CA.

Calder, B., C. Krintz, S. John, and T. Austin. 1998. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. San Jose.

Callahan, D., J. Cocke, and K. Kennedy. 1988a. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550.

Callahan, D., J. Cocke, and K. Kennedy. 1988b. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358.

Carr, S. and K. Kennedy. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810.

Cascaval, G. C. 2000. *Compile-time Performance Prediction of Scientific Programs*. Ph.D. thesis, University of Illinois at Urbana-Champaign.

Chan, B. and T. S. Abdelrahman. 2004. Run-time support for the automatic parallelization of java programs. *Journal of Supercomputing*, 28(1):91–117.

Chandra, D., F. Guo, S. Kim, and Y. Solihin. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

Chen, Michael K. and Kunle Olukotun. 2003. The Jrpm system for dynamically parallelizing java programs. In *30th International Symposium on Computer Architecture*, pages 434–445.

Chen, Trista P., Horst Haussecker, Alexander Bovyrin, Roman Belenov, Konstantin Rodyushkin, Alexander Kuranov, and Victor Eruhimov. 2005. Computer vision workload analysis: Case study of video surveillance systems. *Intel Technology Journal*, 9(2):109–118.

Chilimbi, T. M. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah.

Chilimbi, T. M., B. Davidson, and J. R. Larus. 1999a. Cache-conscious structure definition. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia.

Chilimbi, T. M., M. D. Hill, and J. R. Larus. 1999b. Cache-conscious structure layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia.

Cintra, M. H. and D. R. Llanos. 2005. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576.

Cooper, Keith and Linda Torczon. 2004. *Engineering a Compiler*. Morgan Kaufmann Publishers.

Cytron, R. 1986. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*. St. Charles, IL.

Cytron, R., J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490.

Dang, F., H. Yu, and L. Rauchwerger. 2002. The R-LRPD test: Speculative parallelization of partially parallel loops. Technical report, CS Dept., Texas A&M University, College Station, TX.

Das, R., D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. 1992. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Science Meeting*. Reno, Navada.

Das, R., M. Uysal, J. Saltz, and Y.-S. Hwang. 1994. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479.

Daubechies, I. 1992. *Ten Lectures on Wavelets*. Capital City Press, Montpelier,Vermont.

Denning, P.J. 1980. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1).

Dhodapkar, A. S. and J. E. Smith. 2002. Managing multi-configuration hardware via dynamic working-set analysis. In *Proceedings of International Symposium on Computer Architecture*. Anchorage, Alaska.

Dhodapkar, A. S. and J. E. Smith. 2003. Comparing program phase detection techniques. In *Proceedings of International Symposium on Microarchitecture*.

Ding, C. 2000. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. Ph.D. thesis, Dept. of Computer Science, Rice University.

Ding, C. and K. Kennedy. 1999a. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*. Atlanta, GA.

Ding, C. and K. Kennedy. 1999b. Inter-array data regrouping. In *Proceedings of The 12th International Workshop on Languages and Compilers for Parallel Computing*. La Jolla, California.

Ding, C. and K. Kennedy. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134.

Ding, C., C. Zhang, X. Shen, and M. Ogihara. 2005. Gated memory control for memory monitoring, leak detection and garbage collection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Memory System Performance*. Chicago, IL.

Ding, C. and Y. Zhong. 2003. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA.

Duesterwald, E., C. Cascaval, and S. Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. New Orleans, Louisiana.

Emami, M., R. Ghiya, and L. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

Fang, C., S. Carr, S. Onder, and Z. Wang. 2004. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*. Washington DC.

Fang, C., S. Carr, S. Onder, and Z. Wang. 2005. Instruction based memory distance analysis and its application to optimization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. St. Louis, MO.

Faroughi, Nikrouz. 2005. Multi-cache memory profiling for parallel processing programs. In *The Workshop on Binary Instrumentation and Application*.

Forney, Brian, Steven Hart, and Matt McCornick. 2001. An analysis of cache sharing in chip multiprocessors. (course project report, http://www.cs.wisc.edu/ mattmcc/papers/MPCacheStudy.pdf).

Forum, High Performance Fortran. 1997. High performance fortran language specification, version 2.0. Technical report, CRPC-TR92225, Center for Research on Parallel Computation, Rice University.

Frigo, M., C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Georges, A., D. Buytaert, L. Eeckhout, and K. De Bosschere. 2004. Method-level phase behavior in Java workloads. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*.

Gloy, N. and M. D. Smith. 1999. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5).

Grant, B., M. Philipose, M. Mock, C. Chambers, and S. Eggers. 1999a. An evaluation of staged run-time optimizations in DyC. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, GA.

Grant, B., M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. 1999b. An evaluation of staged run-time optimizations in DyC. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia.

Grossman, Dan. 2006. Software transactions are to concurrency as garbage collection is to memory management. Technical Report UW-CSE 2006-04-01, Dept. of Computer Science and Engineering, University of Washington.

Gupta, M. and R. Nim. 1998. Techniques for run-time parallelization of loops. In *Proceedings of SC'98*.

Halstead, R. H. 1985. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538.

Han, H. and C. W. Tseng. 2000a. Improving locality for adaptive irregular scientific codes. In *Proceedings of Workshop on Languages and Compilers for High-Performance Computing (LCPC'00)*. White Plains, NY.

Han, H. and C. W. Tseng. 2000b. Locality optimizations for adaptive irregular scientific codes. Technical report, Department of Computer Science, University of Maryland, College Park.

Harris, Tim and Keir Fraser. 2003. Language support for lightweight transactions. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. Anaheim, CA.

Havlak, P. and K. Kennedy. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360.

Hendren, L., C. Donawa, M. Emami, G. Gao, Justiani, and B.ăSridharan. 1992. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of LCPC. Lecture Notes in Computer Science No. 457*.

Hennessy, Jone L. and David A. Patterson. 2003. *Computer Architecture: A Quantitative Approach*, chapter 5. Morgan Kaufmann.

Henning, J. 2000. Spec2000: measuring cpu performance in the new millennium. *IEEE Computer*.

Herlihy, M. and J. E. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Co mputer Architecture*. San Diego, CA.

Herlihy, Maurice, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th PODC*, pages 92–101. Boston, MA.

Hopcroft, J. E. and J. D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Addison-Wesley.

Hsu, C.-H. and U. Kremer. 2003. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA.

Huang, M., J. Renau, and J. Torrellas. 2003. Positional adaptation of processors: application to energy reduction. In *Proceedings of the International Symposium on Computer Architecture*. San Diego, CA.

Huang, S. A. and J. P. Shen. 1996. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the 7th International Conferences on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA.

Jeremiassen, T. E. and S. J. Eggers. 1995. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188. Santa Barbara, CA.

Jiang, S. and X. Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Marina Del Rey, California.

Joseph, R., Z. Hu, and M. Martonosi. 2004. Wavelet analysis for microprocessor design: Experiences with wavelet-based di/dt characterization. In *Proceedings of International Symposium on High Performance Computer Architecture*.

Keckler, S.W., D. Burger, C.R. Moore, R. Nagarajan, K. Sankaralingam, V. Agarwal, M.S. Hrishikesh, N. Ranganathan, and P. Shivakumar. 2003. A wire-delay scalable microprocessor architecture for high performance systems. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, pages 1068–1069.

Kejariwal, A., X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. 2006. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of ACM International Conference on Supercomputing*.

Keleher, P., A. Cox, S. Dwarkadas, and W. Zwaenepoel. 1994. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*.

Kennedy, K. and U. Kremer. 1998. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4).

Kim, Seon Wook, Michael Voss, and Rudolf Eigenmann. 1999. Characterization of locality in loop-parallel programs. Technical Report ECE-HPCLab-99201, School of Electrical and Computer Engineering.

Knobe, K. and V. Sarkar. 1998. Array SSA form and its use in parallelization. In *Proceedings of Symposium on Principles of Programming Languages*. San Diego, CA.

Knuth, D. 1971. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133.

Kung, H. T. and J. T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2).

Larus, J. R. 1999. Whole program paths. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia.

Lau, J., E. Perelman, and B. Calder. 2004. Selecting software phase markers with code structure analysis. Technical Report CS2004-0804, UCSD. *conference version to appear in CGO'06*.

Lau, Jeremy, Erez Perelman, and Brad Calder. 2006. Selecting software phase markers with code structure analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*.

Li, K. 1986. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Ph.D. thesis, Dept. of Computer Science, Yale University, New Haven, CT.

Li, Z., J. Gu, and G. Lee. 1996. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjunction with the HPCA-2*. San Jose, California.

Li, Z., P. Yew, and C. Zhu. 1990. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34.

Liu, W. and M. Huang. 2004. Expert: Expedited simulation exploiting program behavior repetition. In *Proceedings of International Conference on Supercomputing*.

Luk, C. and T. C. Mowry. 1999. Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of International Symposium on Computer Architecture*. Atlanta, GA.

Magklis, G., M. L. Scott, G. Semeraro, D. H. Albonesi, , and S. Dropsho. 2003. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the International Symposium on Computer Architecture*. San Diego, CA.

Marathe, Virendra J. and Michael L. Scott. 2004. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, Department of Computer Science, University of Rochester.

Marin, G. and J. Mellor-Crummey. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*. New York City, NY.

Marin, G. and J. Mellor-Crummey. 2005. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamo s Computer Science Institute*. Sante Fe, New Mexico.

Martin, M. K., D. J. Sorin, H. V. Cain, M. D. Hill, and M. H. Lipasti. 2001. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of the International Symposium on Microarchitecture (MICRO-34)*.

Mattson, R. L., J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117.

McKinley, K. 2004. Polar opposites: Next generation languages and architectures. Keynotes Talk on Memory Systems Performance.

McKinley, K. S., S. Carr, and C.-W. Tseng. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453.

McKinley, K. S. and O. Temam. 1999. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336.

Mellor-Crummey, J. 1992. Compile-time support for efficient data race detection in shared memory parallel programs. Technical Report CRPC-TR92232, Rice University.

Mellor-Crummey, J., D. Whalley, and K. Kennedy. 2001. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3).

Merriam-Webster. 1998. *Merriam-Webster's Collegiate Dictionary (10th Edition)*. Merriam-Webster.

Mitchell, N., L. Carter, and J. Ferrante. 1999. Localizing non-affine array references. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. Newport Beach, California.

Moss, J. E. B. 2006. Open nested transactions: semantics and support. In *Proceedings of the Workshop on Memory Performance Issues (WMPI)*.

Nagpurkar, P., M. Hind, C. Krintz, P. F. Sweeney, and V.T. Rajan. 2006. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*.

Nevill-Manning, C. G. and I. H. Witten. 1997. Identifying hierarchical structure in sequences: a linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82.

OpenMP. 2005. OpenMP application program interface, version 2.5. Http://www.openmp.org/drupal/mp-documents/spec25.pdf.

Papadimitriou, C. H. 1979. The serializability of concurrent database updates. *Journal of ACM*, 26(4).

Perkovic, D. and P. J. Keleher. 2000. A protocol-centric approach to on-the-fly race detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1058–1072.

Pingali, V. S., S. A. McKee, W. C. Hsieh, and J. B. Carter. 2003. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31(4).

Rabbah, R. M. and K. V. Palem. 2003. Data remapping for design space optimization of embedded memory systems. *ACM Transactions in Embedded Computing Systems*, 2(2).

Rauchwerger, L. and D. Padua. 1995. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. La Jolla, CA.

Rawlings, J. O. 1988. *Applied Regression Analysis: A Research Tool*. Wadsworth and Brooks.

Rinard, M. C. and M. S. Lam. 1998. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545.

Rivera, G. and C.-W. Tseng. 1998. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

Saltz, J. H., R. Mirchandaney, and K. Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612.

Sarkar, V. 1989. Determining average program execution times and their variance. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Portland, Oregon.

Seidl, M. L. and B. G. Zorn. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. San Jose.

Shavit, Nir and Dan Touitou. 1997. Software transactional memory. *Distributed Computing*, 10(2):99–116.

Shen, X. and C. Ding. 2004. Adaptive data partition for sorting using probability distribution. In *Proceedings of International Conference on Parallel Processing*. Montreal, Canada.

Shen, X. and C. Ding. 2005. Parallelization of utility programs based on behavior phase analysis. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Hawthorne, NY. Short paper.

Shen, X., C. Ding, S. Dwarkadas, and M. L. Scott. 2004a. Characterizing phases in service-oriented applications. Technical Report TR 848, Department of Computer Science, University of Rochester.

Shen, X., Y. Gao, C. Ding, and R. Archambault. 2005. Lightweight reference affinity analysis. In *Proceedings of the 19th ACM International Conference on Supercomputing*. Cambridge, MA.

Shen, X., Y. Zhong, and C. Ding. 2003. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute*. Sante Fe, New Mexico.

Shen, X., Y. Zhong, and C. Ding. 2004b. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*. Boston, MA.

Shen, X., Y. Zhong, and C. Ding. 2004c. Phase-based miss rate prediction. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. West Lafayette, IN.

Sherwood, T., E. Perelman, and B. Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*. Barcelona, Spain.

Sherwood, T., S. Sair, and B. Calder. 2003. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*. San Diego, CA.

Silvera, R., R. Archambault, D. Fosbury, and B. Blainey. unpublished. Branch and value profile feedback for whole program optimization. Unpublished, no date given.

Smith, Alan J. 1978. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4:121–130.

So, B., M. W. Hall, and P. C. Diniz. 2002. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Berlin, Germany.

Sohi, G. S., S. E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*.

Srivastava, A. and A. Eustace. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. Orlando, Florida.

Steffan, J. G., C. Colohan, A. Zhai, and T. C. Mowry. 2005. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300.

Strout, M. M., L. Carter, and J. Ferrante. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA.

Sugumar, R. A. and S. G. Abraham. 1993. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*. Santa Clara, CA.

Sugumar, Rabin A. and Santosh G. Abraham. 1991. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan.

Thabit, K. O. 1981. *Cache Management by the Compiler*. Ph.D. thesis, Dept. of Computer Science, Rice University.

Triolet, R., F. Irigoin, and P. Feautrier. 1986. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. Palo Alto, CA.

Voss, M. and R. Eigenmann. 2001. High-level adaptive program optimization with adapt. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*. Snowbird, Utah.

Wagner, T. A., V. Maverick, S. L. Graham, and M. A. Harrison. 1994. Accurate static estimators for program optimization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

Wahbe, R., S. Lucco, and S. L. Graham. 1993. Practical data breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Albuquerque, NM.

Wegman, M. and K. Zadeck. 1985. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*. New Orleans, LA.

Whaley, R. C., A. Petitet, and J. Dongarra. 2001. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2).

Wolf, M. E. and M. Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*. Toronto, Canada.

Zhang, Chengliang, Kirk Kelsey, Xipeng Shen, and Chen Ding. 2006. Program-level adaptive memory management. In *ISMM '04: Proceedings of the 2006 International Symposium on Memory Management*. Ottawa, Canada.

Zhang, L. 2000. *Efficient Remapping Mechanism for an Adaptive Memory System*. Ph.D. thesis, Department of Computer Science, University of Utah.

Zhang, L., Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. 2001. The Impulse memory controller. *IEEE Transactions on Computers*, 50(11).

Zhong, Y. 2005. *Distance-based Whole-Program Data Locality Hierarchy*. Ph.D. thesis, University of Rochester.

Zhong, Y., C. Ding, and K. Kennedy. 2002. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. Washington DC.

Zhong, Y., S. G. Dropsho, and C. Ding. 2003a. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. New Orleans, Louisiana.

Zhong, Y, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. To appear. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers (TOC)*.

Zhong, Y., M. Orlovich, X. Shen, and C. Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Zhong, Y., X. Shen, and C. Ding. 2003b. A hierarchical model of reference affinity. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*. College Station, Texas.

Zhou, Y., P. M. Chen, and K. Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Technical Conference*.